

```
#pragma omp parallel for num_threads(nbt)
for (int i=0; i<m; i++)
```

# Utiliser git

Romain BOMAN

```
int idx2=0;
for(int nbt=trange.getMin(); nbt<=trange.getMax(); nbt+=trange.getStep())
{
    idx2++;
    double tstart = omp_get_wtime();
    test.execute(nbt);
    double tstop = omp_get_wtime();
```

version 22 - 10 août 2023



# Utiliser git – table des matières

- Introduction
- Installation/configuration
- PARTIE 1: Utiliser git en local
  - Les bases
  - Je nettoie mon travail
  - Derniers mots
- PARTIE 2: Utiliser git avec un remote
  - Configurer GitLab
  - Cloner un projet git
  - Une seule branche sur origin
  - Plusieurs branches sur origin
- PARTIE 3: En pratique
  - Utiliser git à travers un IDE – VS Code
  - Merge requests, issues, CI pipelines
- Divers

# Utiliser git

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int tmax)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        #pragma omp parallel for num_threads(nbt)
```

## Introduction

```
double tstart = omp_get_wtime();
test.execute(nbt);
double tstop = omp_get_wtime();
double cpu = tstop-tstart;

OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

std::cout << res;
```



# Qu'est ce que git?

Système de **gestion de versions** (VCS – Version Control System)

- SCM system – Source Code Management system.
- Permet de garder trace des différentes versions d'un ensemble de fichiers (textes).
  - Qui à modifié tel fichier? Quand? Comment? Pourquoi?
- Permet de comparer ces versions
  - Facilite la recherche de bug par comparaison avec une version non buguée.
- Facilite/automatise la fusion de versions
  - Version unique de référence.
- Sert de backup de son travail.

Créé initialement par Linus Torvalds (Linux).

Autres systèmes similaires:

- subversion (SVN), mercurial, CVS, etc.

Interfaces web (github, GitLab, bitbucket, etc.)



# Pourquoi git comme VCS?

## ***Avantages***

- gestion décentralisée -> permet de travailler "offline".
- très efficace pour la fusion de versions ("merge").
- interfaces web conviviales (github, etc).
- très populaire (un problème? -> demande à Google).

## ***Inconvénients***

- complexe par rapport à d'autres VCS (branches inévitables, milliers d'options).
- moins intuitif que subversion:
  - "staging index",
  - numéros de version: "hash",
  - plusieurs commandes pour le même effet.
- plusieurs façons de travailler sont possibles ( "git workflows" ).
- l'arbre de versions peut devenir vite très confus!

Il est nécessaire de définir une stratégie adaptée à la situation.



# Pourquoi ce n<sup>ième</sup> tuto?

Cette présentation...

- suit un projet créé à partir de 0,
- part du simple vers le compliqué,
- introduit un **nombre limité** de commandes pour être à l'aise avec git tout en tirant parti de ses avantages (les branches, par exemple),
- présente les **pièges** à éviter,
- définit une manière simple de travailler (**workflow**) pour un collaboration optimale avec des "non-geeks",
- utilise les **outils ULiège** ( <https://gitlab.uliege.be/> ),
- fait un lien avec les vieux outils (**SVN**),
- privilégie **Windows**.



# Pourquoi ce n<sup>ième</sup> tuto?

## Icones:

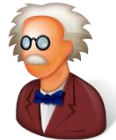
Vu la quantité d'information de ce document, des icônes peuvent aider le lecteur à "sauter" différentes parties de la présentation en fonction de son niveau git initial :



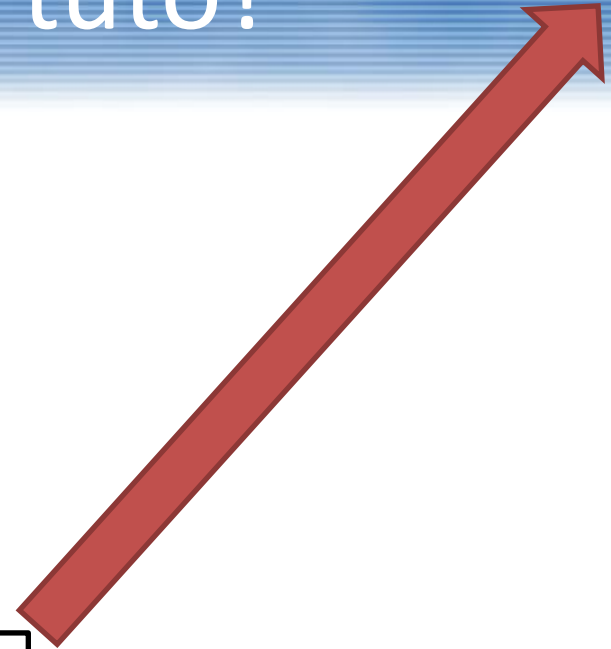
Débutant



Expert



Utilisateur SVN





# Utiliser git

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int tmax)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        #pragma omp parallel for num_threads(nbt)
```

## Installation/configuration

```
double tstart = omp_get_wtime();
test.execute(nbt);
double tstop = omp_get_wtime();
double cpu = tstop-tstart;

OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

std::cout << res;
```

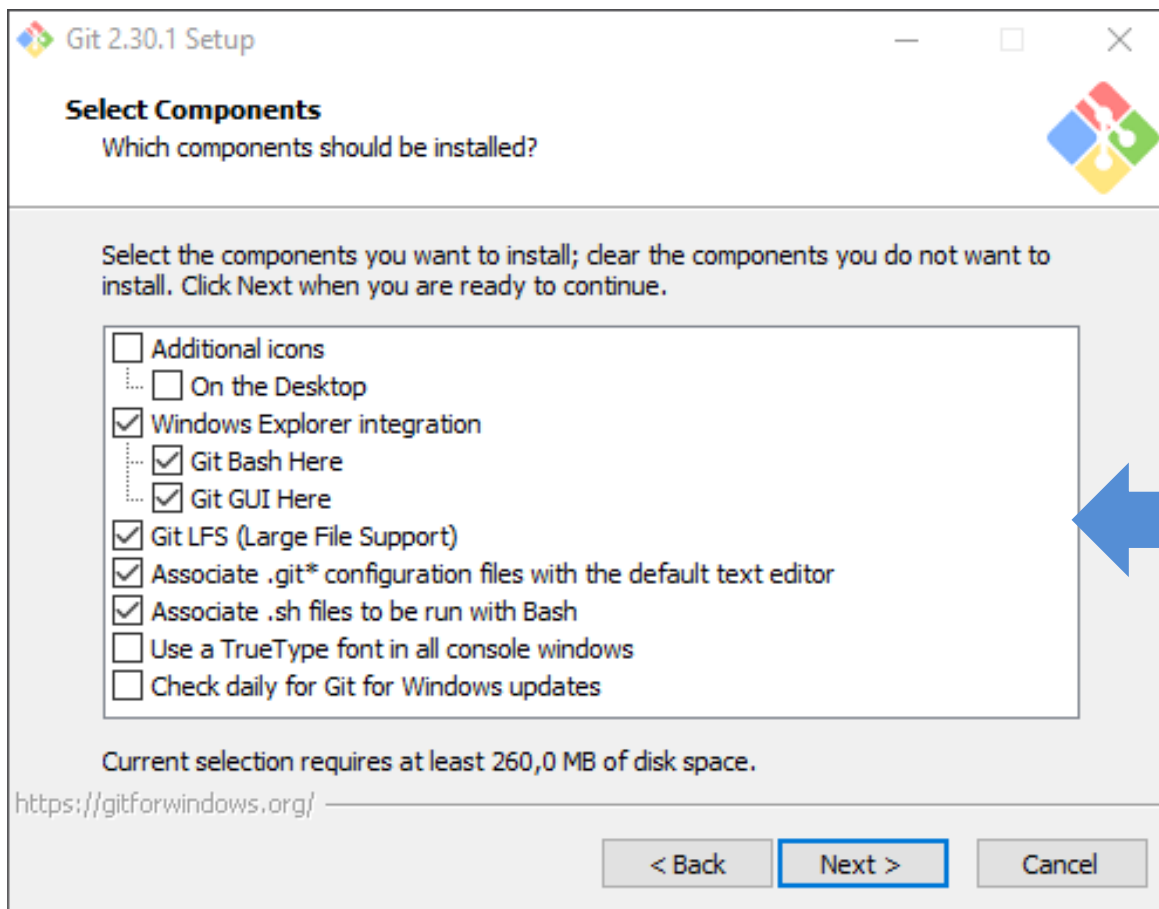




# Installer/configurer git

Windows:

- site officiel: <https://git-scm.com/>

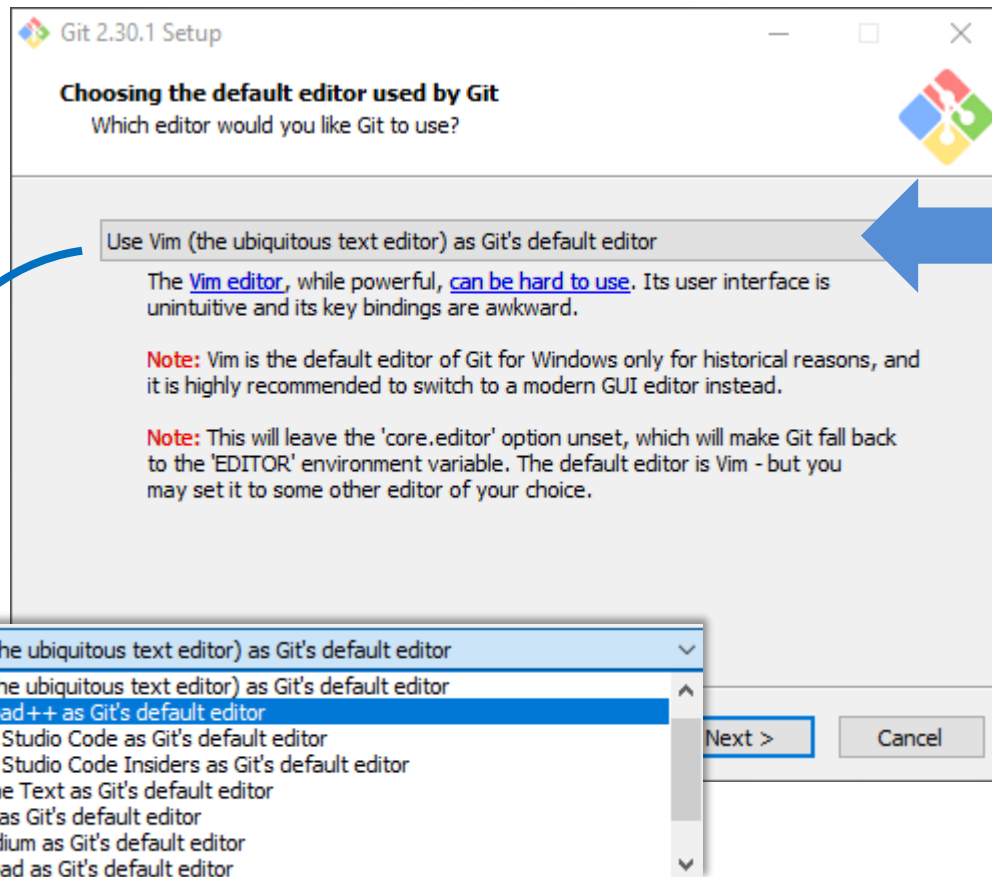


Laisser les  
options par  
défaut



# Installer/configurer git

Même en ligne de commande, git requiert un éditeur de texte



Vous devez choisir ici un éditeur LEGER que vous connaissez bien pour interagir avec git  
(Notepad++ p. expl.)

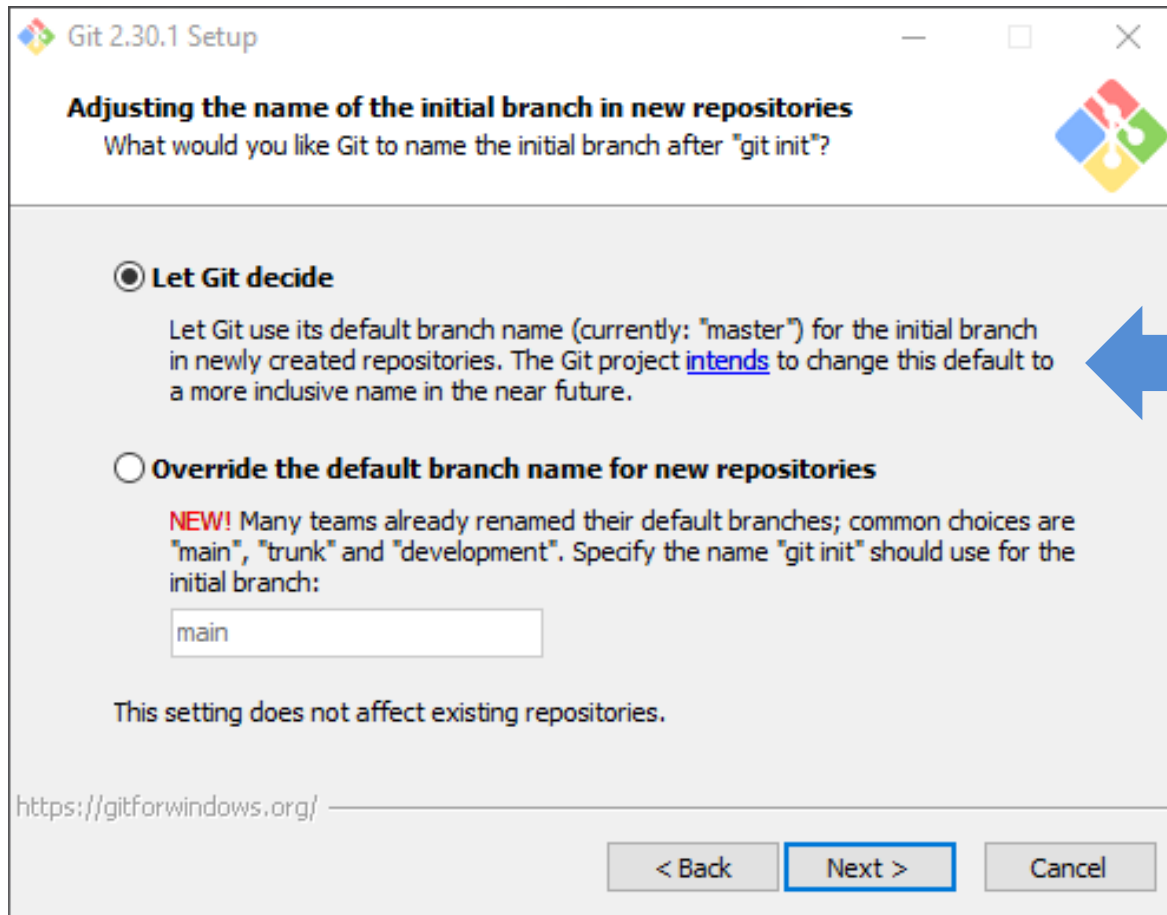
Il ne s'agit pas de l'éditeur avec lequel vous allez programmer.

Sachez que "vim" est disponible sur toutes les machines, c'est peut-être l'occasion d'apprendre à s'en servir (attention, ce n'est pas simple!)

Sous Linux, git utilise l'éditeur défini par la variable d'environnement EDITOR



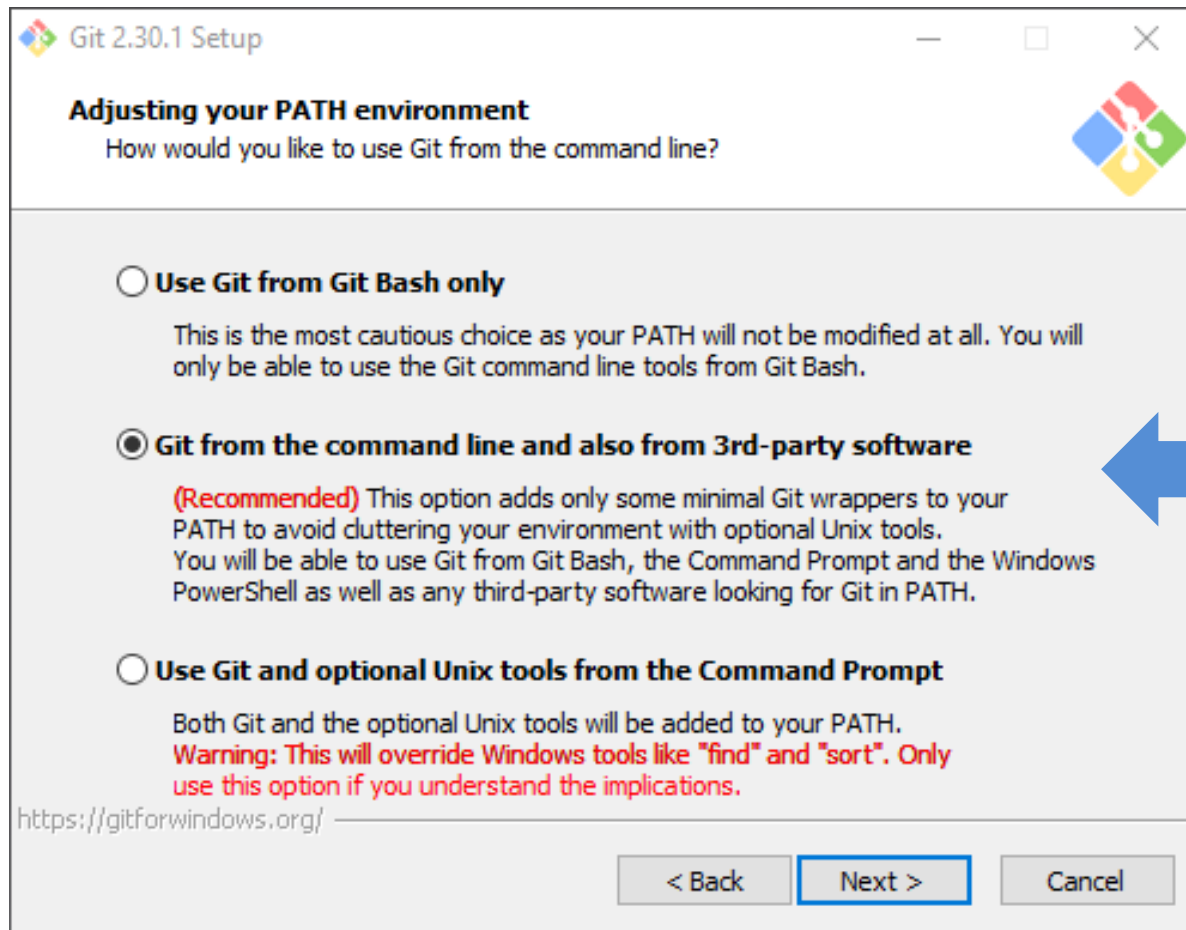
# Installer/configurer git



Laisser l'option par défaut



# Installer/configurer git

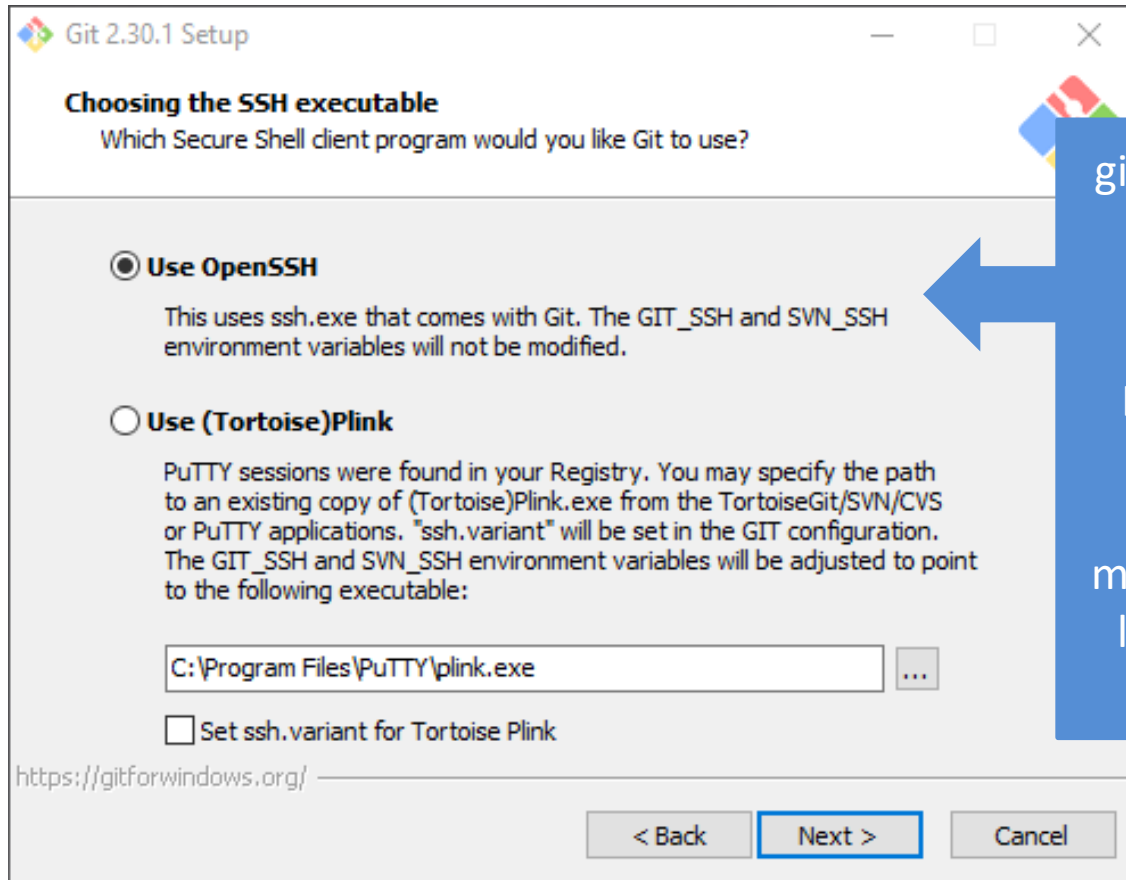


Laisser l'option par défaut

Cette option ajoute au PATH  
système le chemin  
C:\Program Files\Git\cmd



# Installer/configurer git



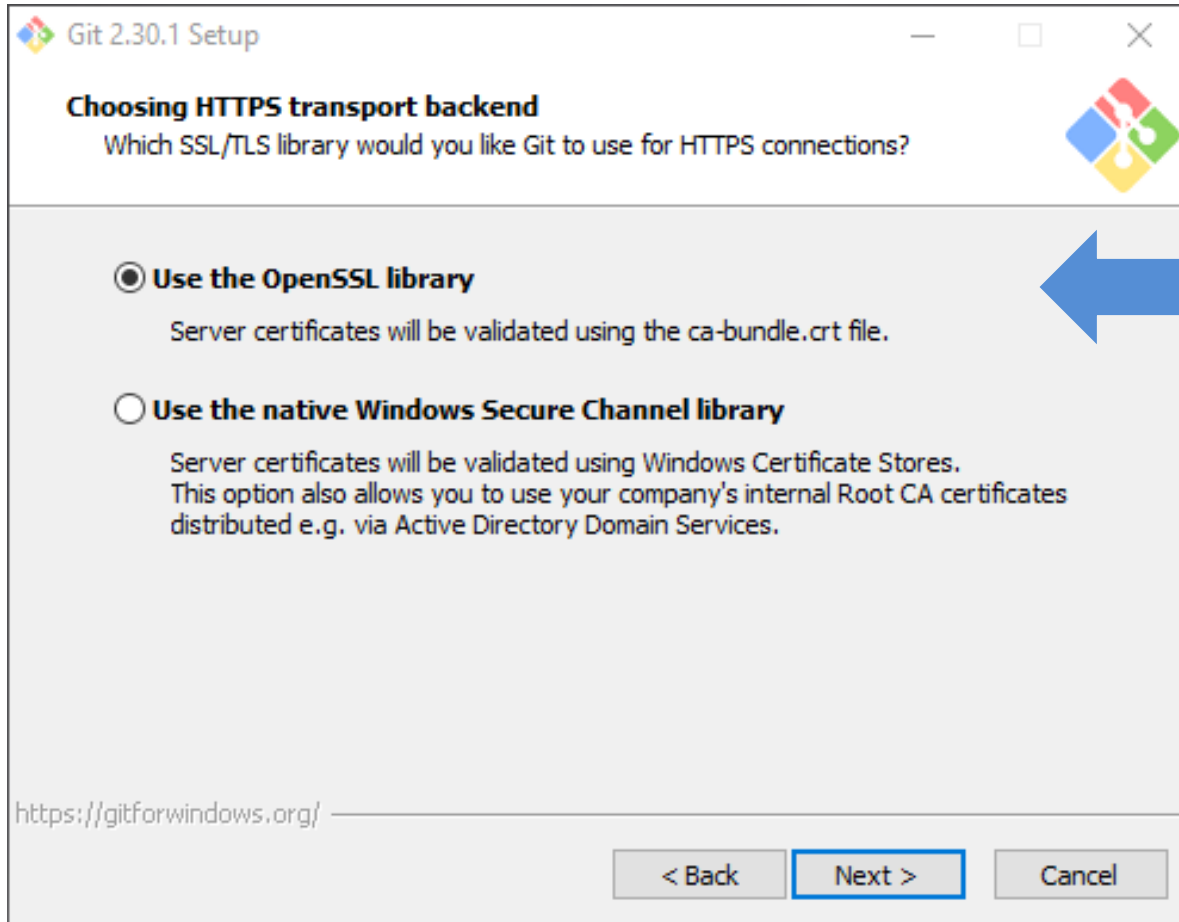
git va utiliser SSH pour se connecter à des machines distantes (voir partie 2).

Le plus simple est d'utiliser le ssh fourni avec git (OpenSSH)

mais vous pouvez également utiliser le client SSH de Tortoise et PUTTY (voir Divers)



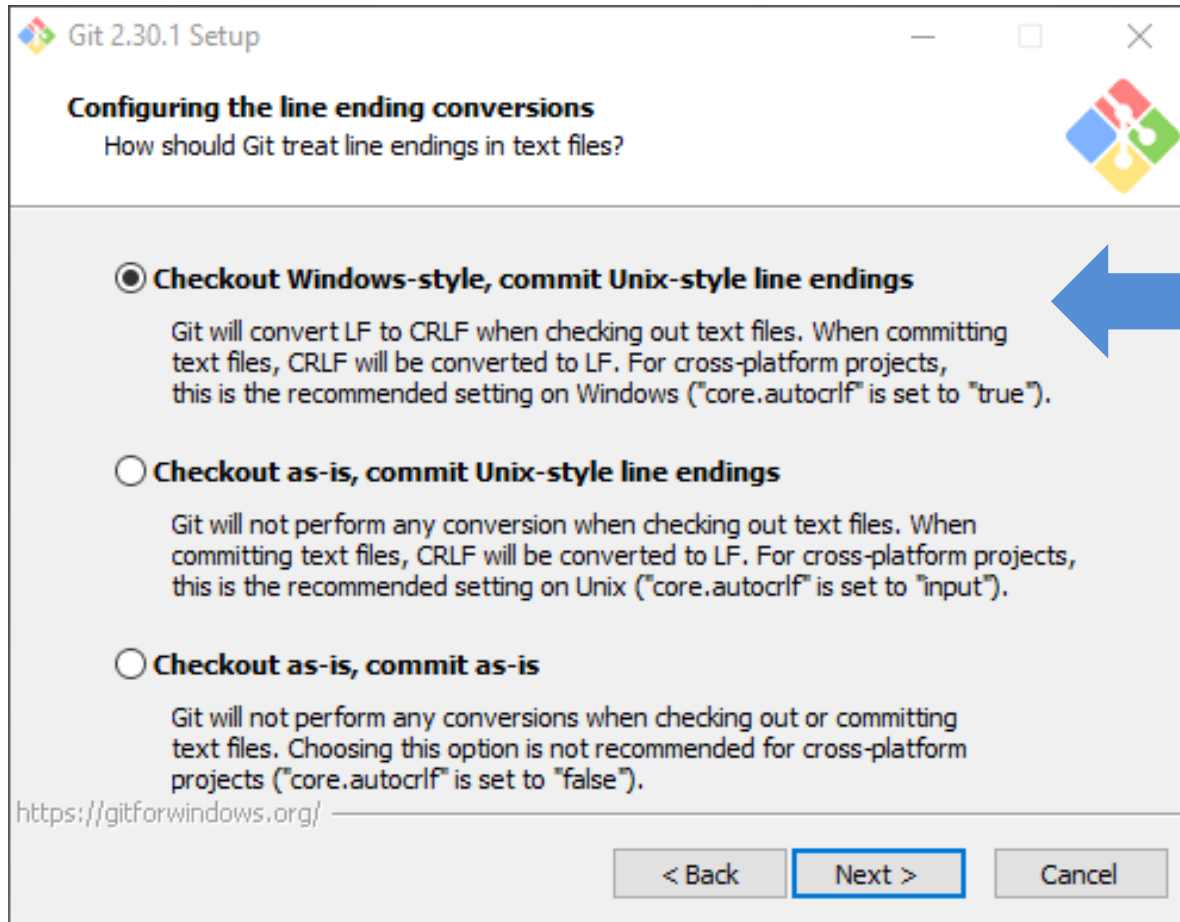
# Installer/configurer git



Laisser  
l'option  
par défaut



# Installer/configurer git



Laisser l'option par défaut

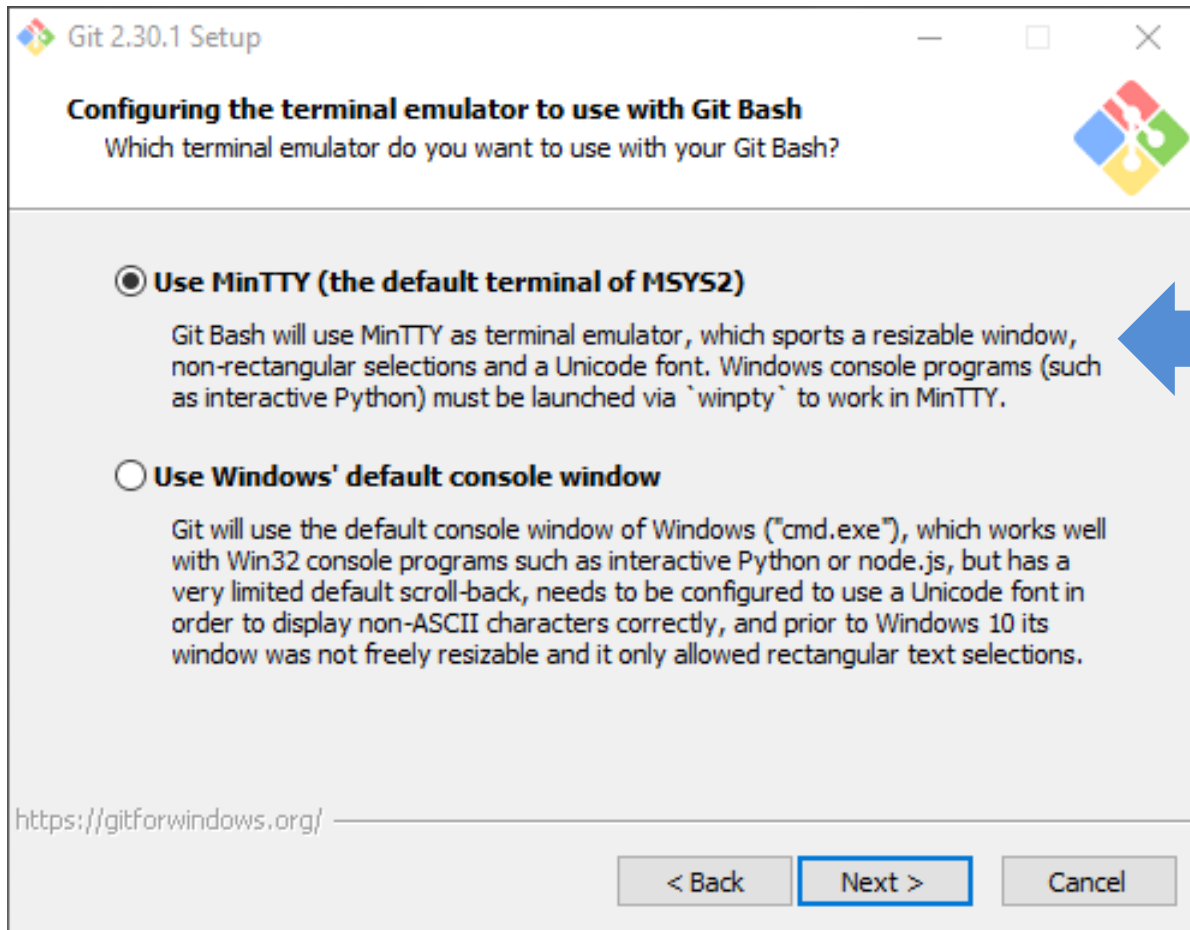
La plupart des serveurs git (gitlab, github, etc.) utilisent par défaut des fins de ligne Unix (\n) et non Windows (\r\n).

Cette option permet d'éviter de mélanger les fins de ligne Windows et Unix lorsqu'on travaille avec les 2 systèmes.





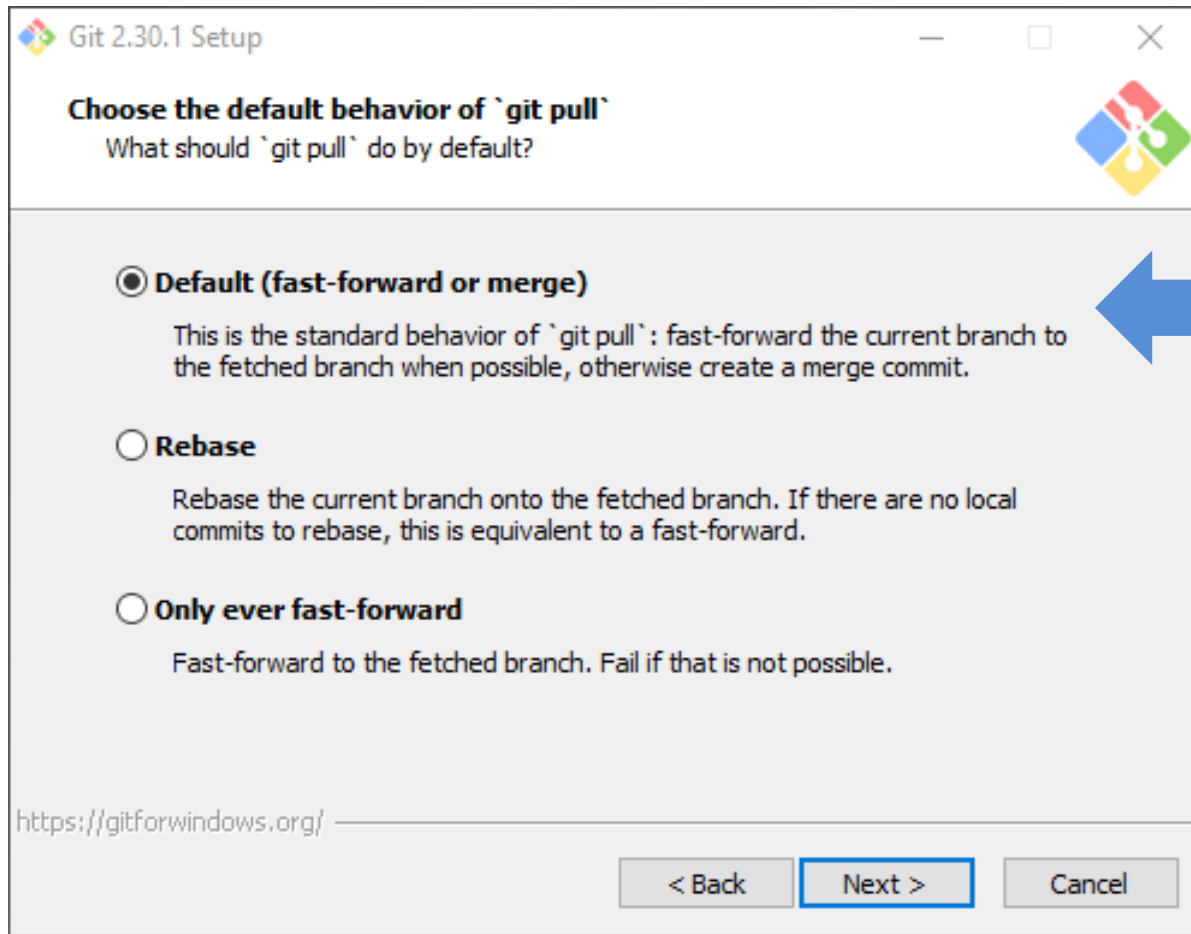
# Installer/configurer git



Laisser  
l'option  
par défaut



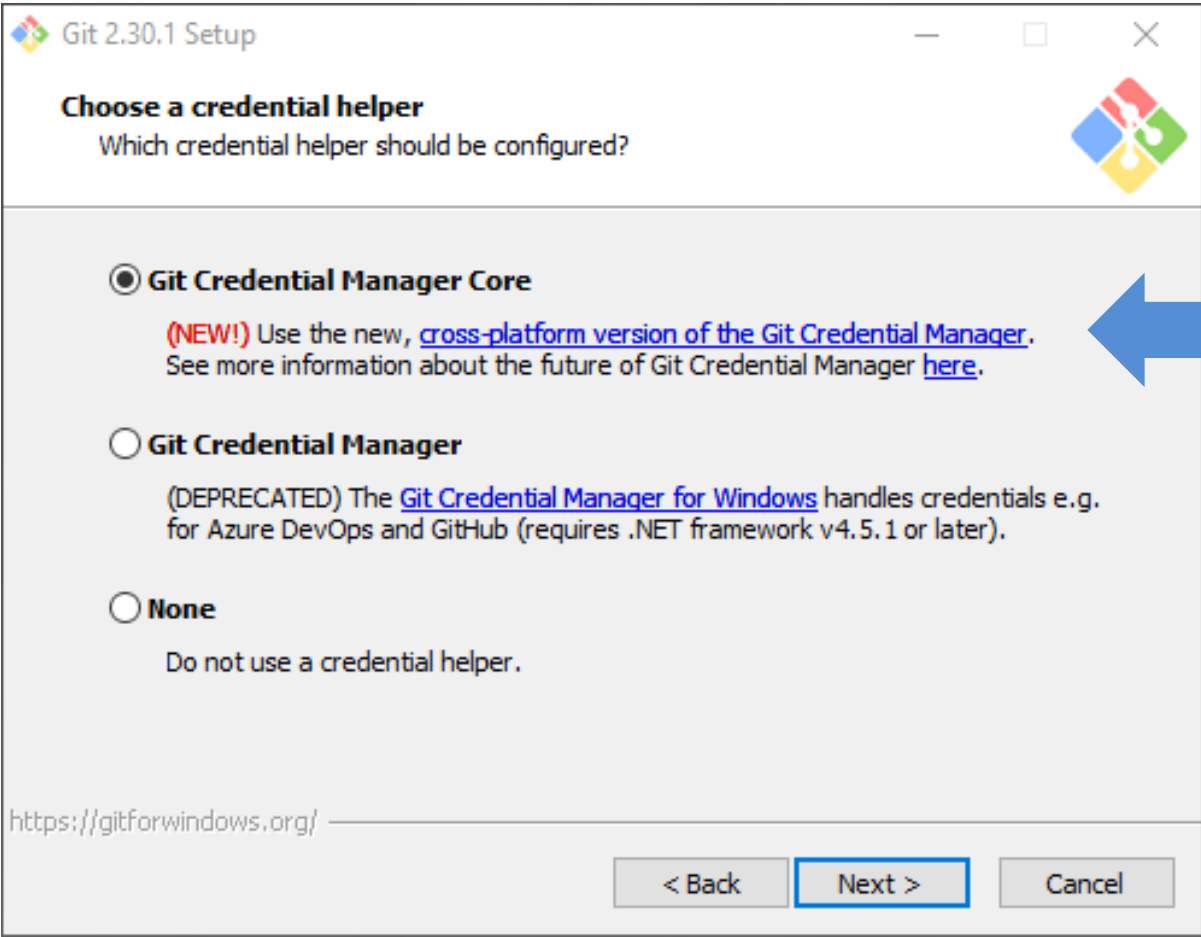
# Installer/configurer git



Laisser  
l'option  
par défaut



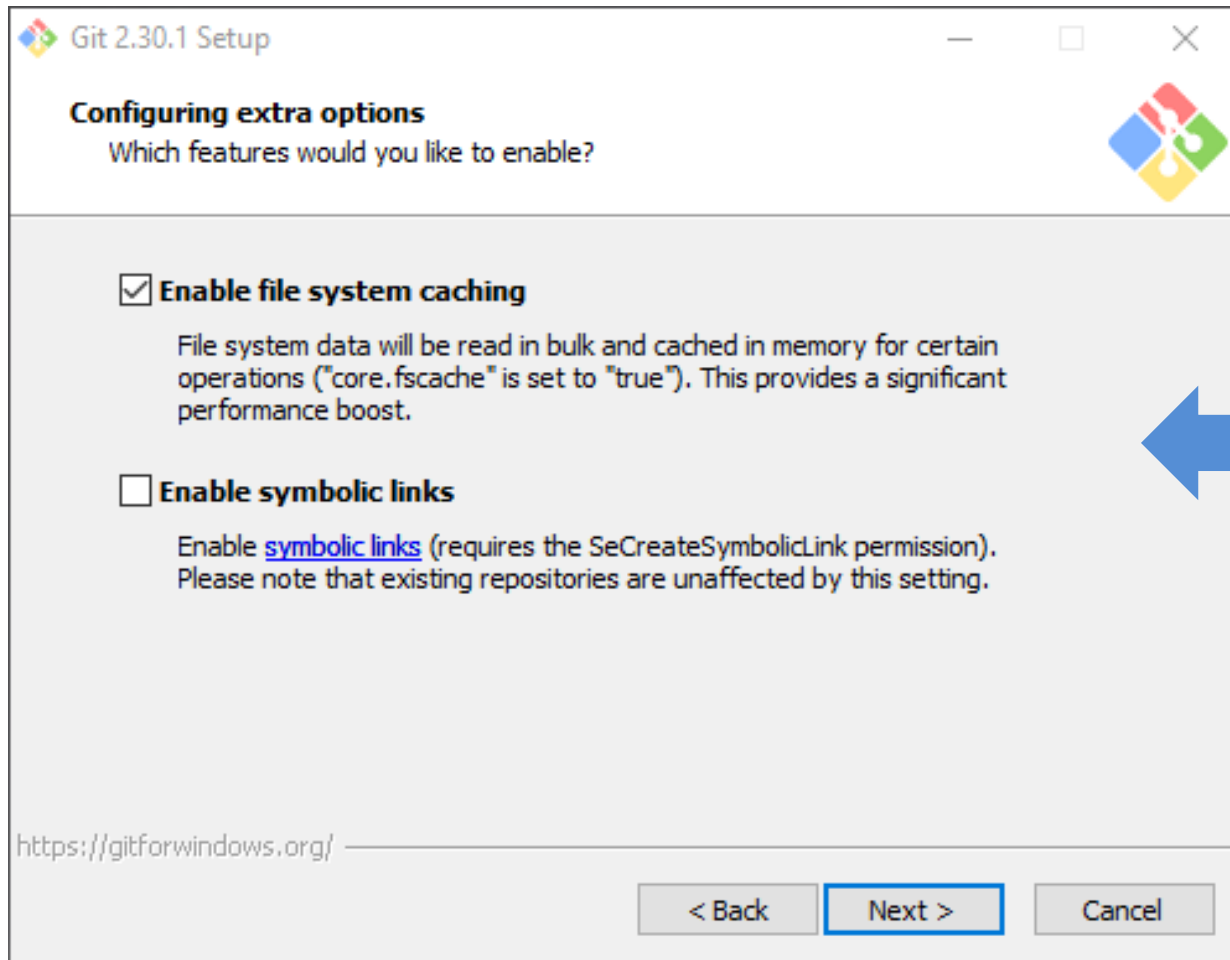
# Installer/configurer git



Laisser l'option par défaut



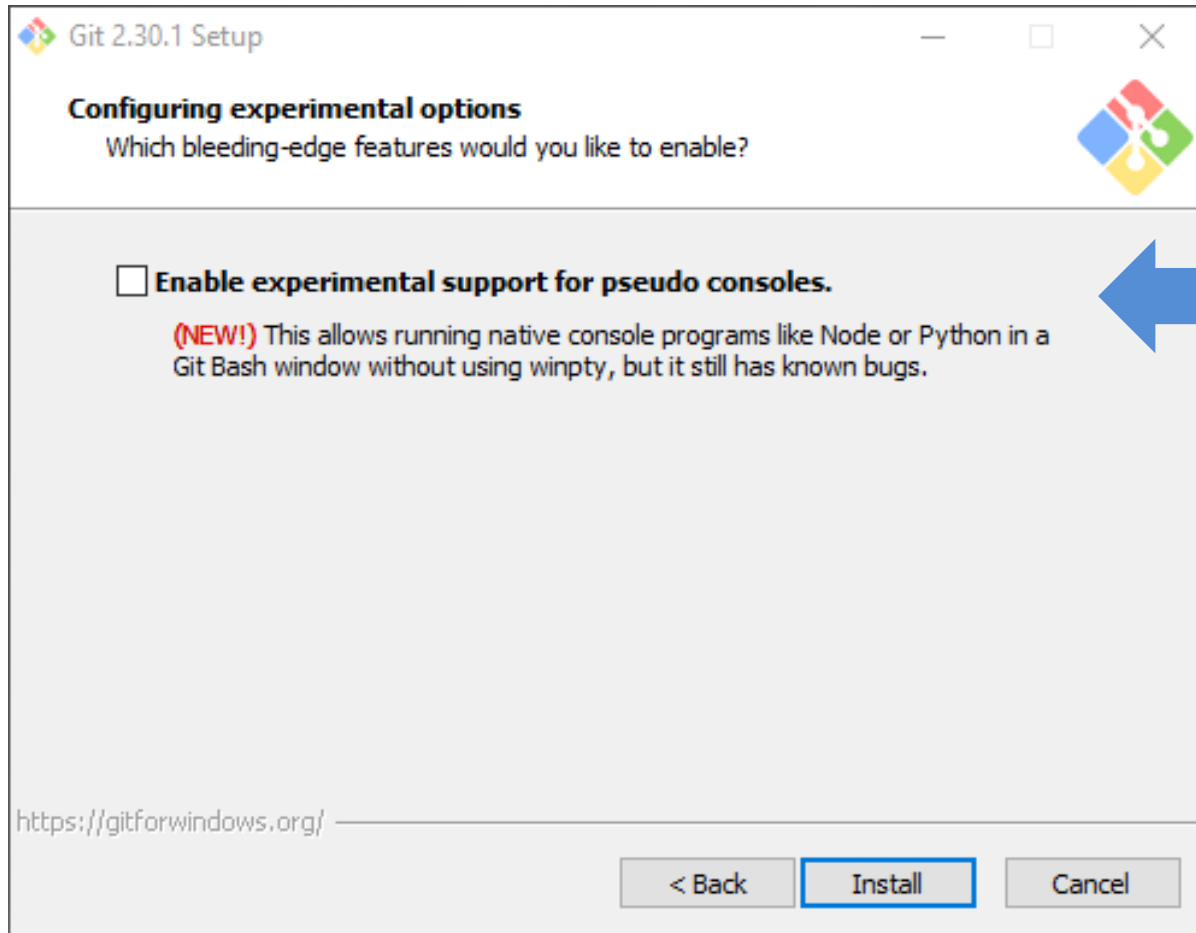
# Installer/configurer git



Laisser les  
options par  
défaut



# Installer/configurer git



Ne pas  
toucher à  
ça



# Installer/configurer git

Qu'est ce qui est installé? et où?



This PC > Local Disk (C:) > Program Files > Git				
Name	Date modified	Type	Size	
bin	22/01/2018 09:50	File folder		
cmd	22/01/2018 09:50	File folder		
dev	22/01/2018 09:51	File folder		
etc	22/01/2018 09:51	File folder		
mingw64	22/01/2018 09:50	File folder		
tmp	22/01/2018 09:51	File folder		
usr	22/01/2018 09:51	File folder		
git-bash.exe	18/01/2018 21:12	Application	145 KB	
git-cmd.exe	18/01/2018 21:12	Application	144 KB	
LICENSE.txt	18/01/2018 07:16	Text Document	19 KB	
ReleaseNotes.html	18/01/2018 21:23	Chrome HTML Do...	117 KB	
setup.ini	22/01/2018 09:51	Configuration sett...	1 KB	
unins001.dat	22/01/2018 09:51	DAT File	1089 KB	
unins001.exe	22/01/2018 09:50	Application	1260 KB	
unins001.msg	22/01/2018 09:51	Élément Outlook	23 KB	

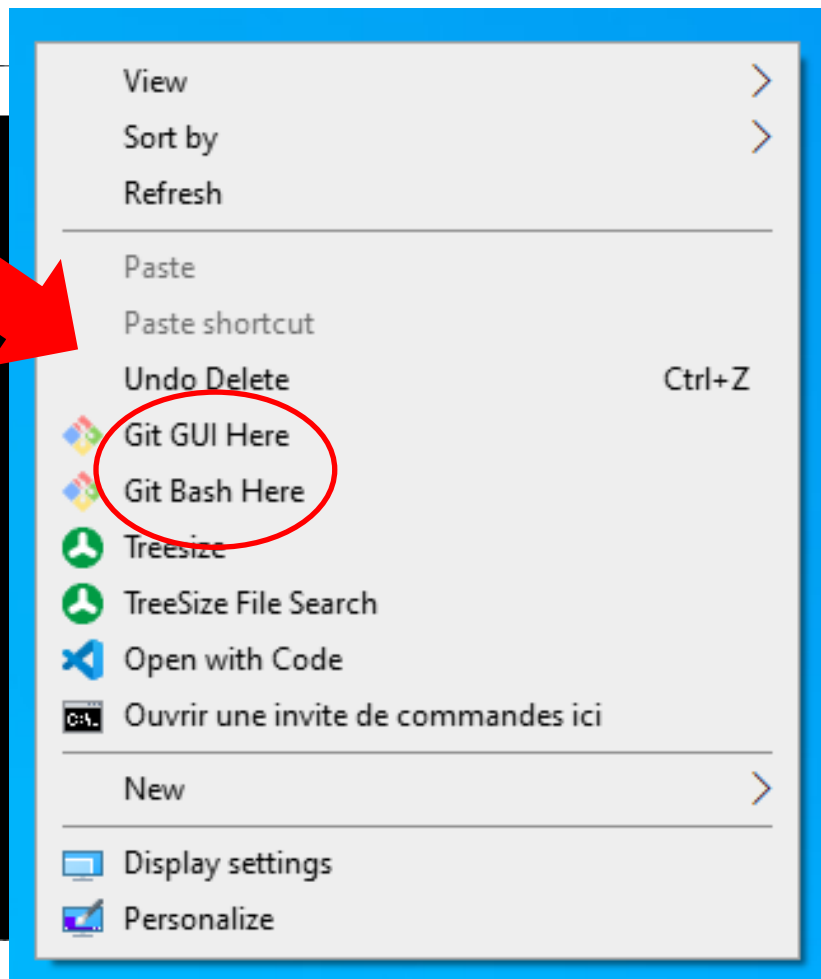
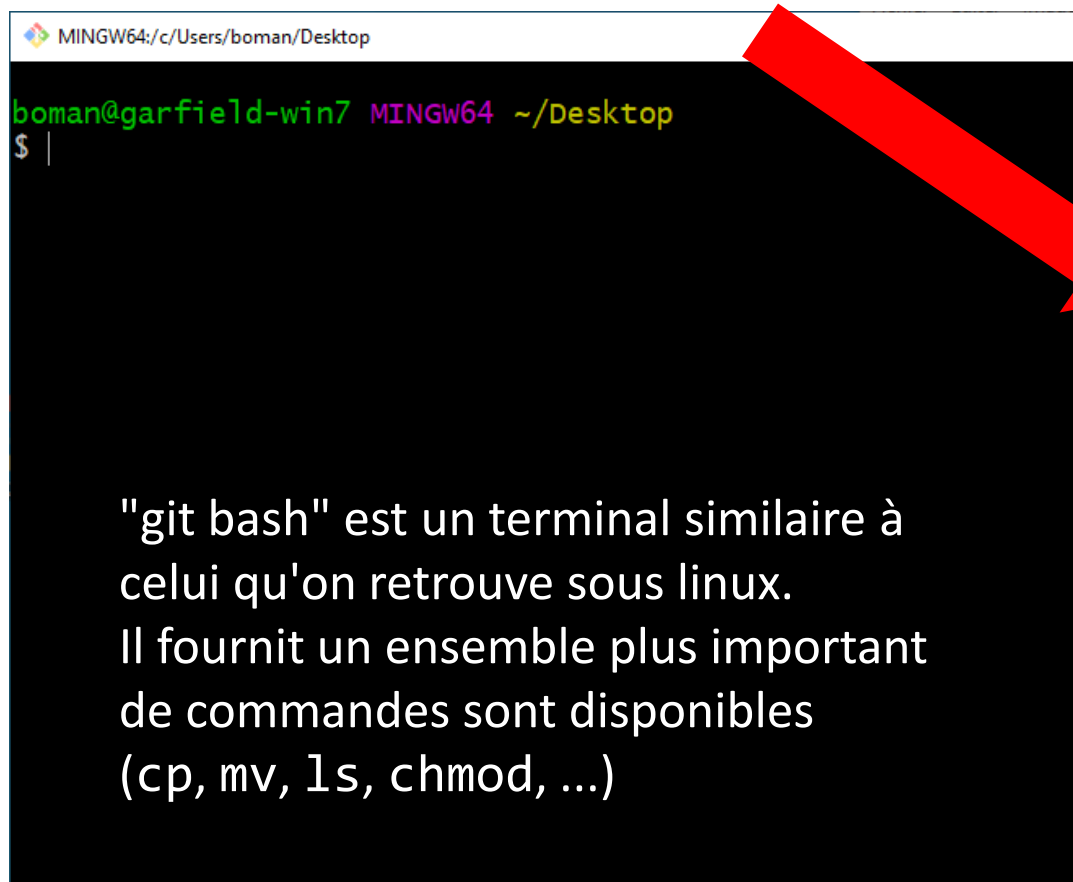
git.exe
git-gui.exe
gitk.exe
git-lfs.exe
start-ssh-agent.cmd
start-ssh-pageant.cmd

Ces commandes  
sont accessibles  
depuis tout  
terminal Windows.



# Installer/configurer git

git est aussi utilisable via "**Git Bash Here**" dans le menu contextuel (**clic droit** dans un dossier ou dans un dossier)



"git bash" est un terminal similaire à celui qu'on retrouve sous linux. Il fournit un ensemble plus important de commandes sont disponibles (cp, mv, ls, chmod, ...)





# Installer/configurer git

## Configuration de git (1x par machine)

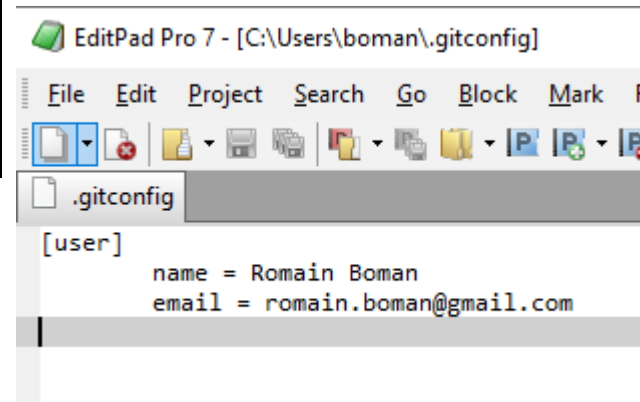
```
git config --global user.name "Your Name"  
git config --global user.email "your_email@uliege.be"
```

MINGW64:/c/Users/boman/Desktop

```
boman@garfield-win7 MINGW64 ~/Desktop  
$ git config --global user.name "Romain Boman"  
  
boman@garfield-win7 MINGW64 ~/Desktop  
$ git config --global user.email "romain.boman@gmail.com"  
  
boman@garfield-win7 MINGW64 ~/Desktop  
$ |
```

La config de git est écrite dans un fichier `.gitconfig` dans votre dossier utilisateur `C:\Users\xxxxx`

`git config -l`  
permet de voir toutes les options actives

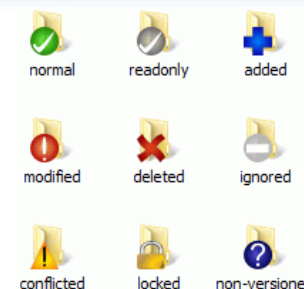




# Installer/configurer git

## *Note destinée aux utilisateurs de SVN*

...et TortoiseGIT ?



Alors que TortoiseSVN peut être utilisé exclusivement pour gérer un projet sous SVN, TortoiseGIT **n'est pas conseillé** pour effectuer des commandes git au delà de simples synchronisations: git est **beaucoup plus complexe que SVN** et vous risquez de faire des opérations non voulues!

La seule utilité de TortoiseGIT est la fonction "**overlay**" dans l'explorateur Windows qui peut vous permettre de voir rapidement l'état de votre travail.

### Conséquence:

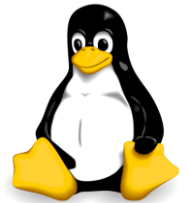
- Contrairement à SVN, pour git, il est impératif de se familiariser d'abord avec la ligne de commande!
- Il est ensuite possible d'utiliser les outils intégrés aux IDEs tels que Visual Studio Code.
- Les sites comme github, GitLab proposent également des aides sous forme d'interfaces web pour naviguer dans un projet git.



# Installer/configurer git

## ***Sous Linux***

```
sudo apt install git
```



note: sous macOS, git est installé par défaut avec le système.

écrit dans  
~/.gitconfig

## ***Configuration (1 seule fois/machine)***

```
git config --global user.name "Your Name"  
git config --global user.email "your_email@uliege.be"  
  
git config --global core.autocrlf input
```

Facultatif:  
les fichiers qui viendraient de Windows  
avec des fins de ligne non-Unix ne  
seront pas considérés "modifiés".

# Utiliser git

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int tmax)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            c[i*n+j] = a[i*n+j] + b[i*n+j];
}
```

## PARTIE 1

# Utiliser git en local

```
double tstop = omp_get_wtime();
double cpu = tstop-tstart;

OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

std::cout << res;
```

# Utiliser git

## PARTIE 1 Utiliser git en local

### On apprend les bases

seul, sur sa machine, en ligne de commande, sans réseau, sans GitLab



# git en local - les bases

## *Gestion d'un répertoire (nommé "code") sur sa machine*

Ouvrir git bash, et taper les commandes suivantes:

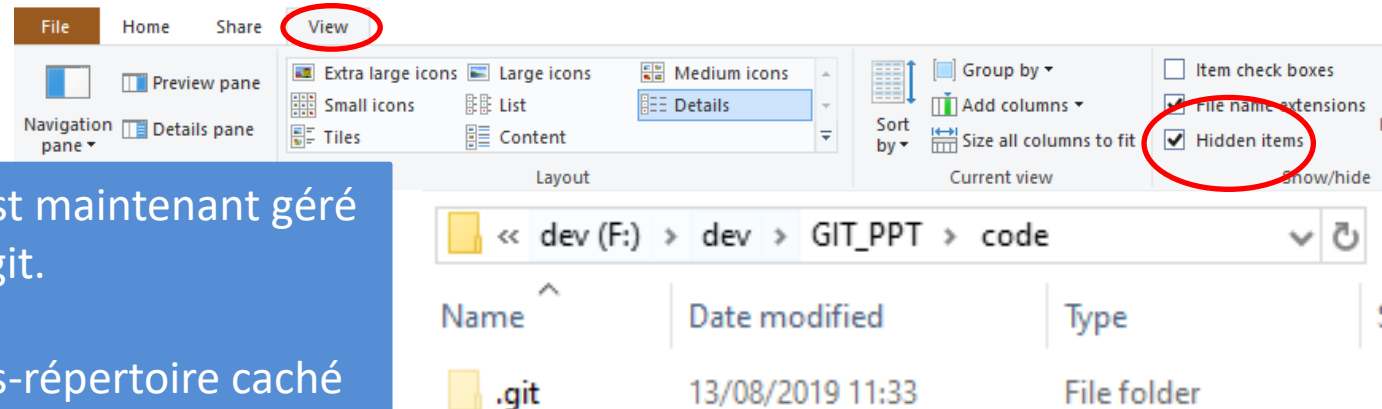
```
mkdir code  
cd code  
  
git init
```



Le répertoire code est maintenant géré par git.

En pratique, un sous-répertoire caché code/.git/ a été créé.

Il ne doit **jamais** être modifié/supprimé!





# git en local - les bases

**1<sup>ère</sup> commande à retenir et à exécuter sans cesse:**

« Quel est l'état de mon répertoire? » : `git status`

Git peut gérer simultanément plusieurs variantes de mon code (**branches**).  
Pour l'instant il n'y a qu'une branche.  
La branche par défaut s'appelle "master"

`git status`

On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)

A ce stade, un **commit** est synonyme de "version" (ou "révision").  
Pour l'instant, on n'a pas créé de version du code. Tout est vide.

De plus, le répertoire de travail est vide.  
Il n'y a donc pas de possibilité de créer une nouvelle version du code.

Toujours bien lire ce que suggère git.  
C'est généralement très pertinent.





# git en local - les bases

**Créer un fichier.** Par exemple:

```
vi solver.py  
... [écrire du code]
```

Utilisez votre éditeur favori  
(Notepad++, EditPad Pro, Atom, etc.)

...et à nouveau: "git status" ...et lire!

```
git status
```

```
On branch master
```

```
No commits yet
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
    solver.py
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Le fichier `solver.py` est "untracked".  
C'est un fichier que git ne connaît pas.  
Git nous suggère de l'ajouter à la liste  
des fichiers qu'il connaît avec la  
commande "git add".



# git en local - les bases

**"Ajouter" le fichier** (aux fichiers gérés par git)

```
git add solver.py
```

...et à nouveau: "git status" ...et lire!

```
git status
```

```
On branch master
```

```
No commits yet
```

```
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)
```

```
new file:  solver.py
```

git nous dit qu'il est prêt à créer une nouvelle version (un « commit »)

git nous donne la commande qu'il faudrait exécuter pour retourner à l'état précédent, c'est-à-dire supprimer le "git add"

dev (F:) > dev > GIT_PPT > code			▼	🔄
Name ^	Date modified	Type		
.git	13/08/2019 11:46	File folder		
solver.py	13/08/2019 11:43	Python File		



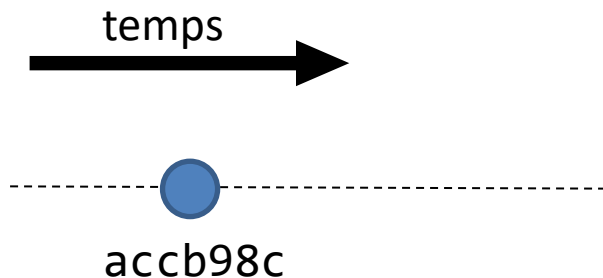
# git en local - les bases

## Créer un 1<sup>er</sup> commit:

Message de commit: explique ce qui a été ajouté/modifié. Très important!

```
git commit -m "add file solver.py"
[master (root-commit) accb98c] adding file solver.py
1 file changed, 2 insertions
create mode 100644 solver.py
```

Identifiant du commit (unique, créé par git – parfois appelé « **commit hash** » – dépend de tout ce qui constitue le commit: mods, date, nom de l'auteur, etc.)



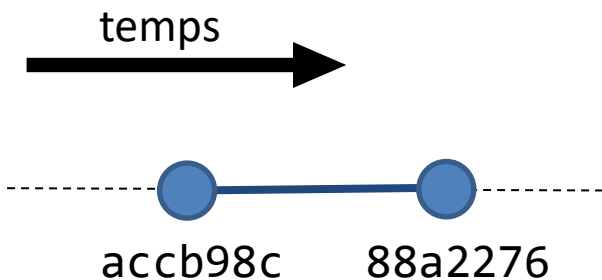
Chaque commit (ou "version", ou "état du code" à un moment donné) va être représenté par **une petite boule** sur une **ligne de temps**



# git en local - les bases

**Modifier le fichier et créer un 2e commit:**

```
vi solver.py
... [modifier le code]
git add solver.py
git commit -m "add python header"
[master 88a2276] add python header
1 file changed, 2 insertions(+)
```



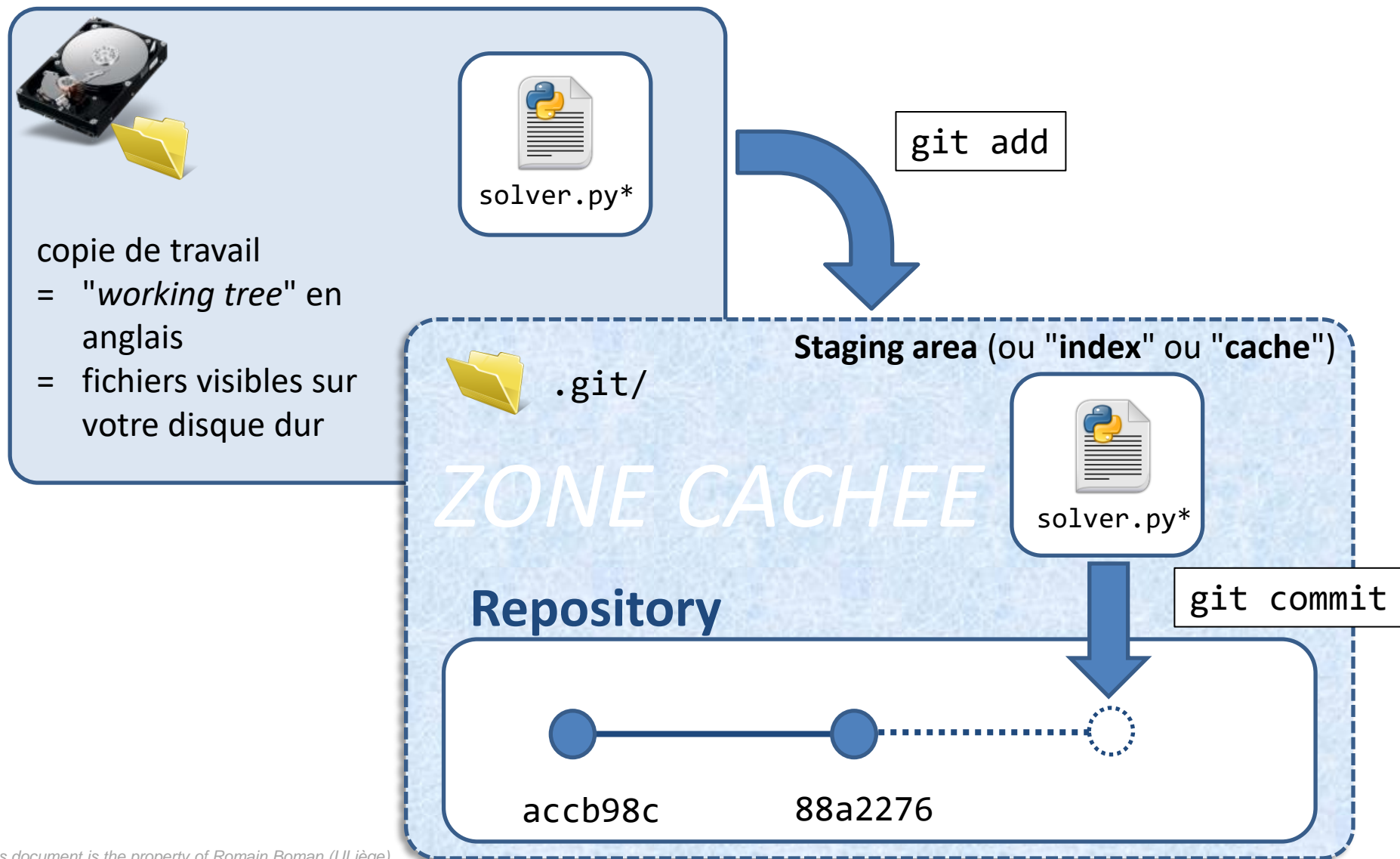
Nouvelle "boule" dans la représentation graphique

Ces versions sont stockées dans une base de données appelée le **repository**



# git en local - les bases

Mécanisme en 2 temps, **pas très intuitif**:

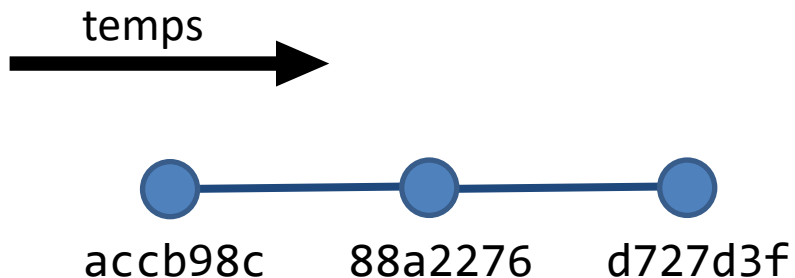




# git en local - les bases

**Modifier le fichier et créer un 3e commit:**

```
vi solver.py
... [modifier le code]
git add solver.py
git commit -m "better output"
[master d727d3f] better output
1 file changed, 1 insertion(+), 1 deletion(-)
```



Nouvelle "boule" dans la  
représentation graphique  
du repository



# git en local - les bases

## Voir l'historique

```
git log
```

```
commit d727d3ff8be35331a857acb326ecd9928aedff0c (HEAD -> master)
Author: Romain Boman <romain.boman@gmail.com>
Date:   Tue Aug 13 14:31:56 2019 +0200
```

```
    better output
```

```
commit 88a2276b527f22b696fa20bf5e754bb9ff0b898a
Author: Romain Boman <romain.boman@gmail.com>
Date:   Tue Aug 13 14:27:52 2019 +0200
```

```
    add python header
```

```
commit accb98cb7285fa2157943a2653c95e1ef1045421
Author: Romain Boman <romain.boman@gmail.com>
Date:   Tue Aug 13 11:46:46 2019 +0200
```

```
    add file solver.py
```

L'identifiant complet de commit est très long, seuls les 1ers caractères peuvent être utilisés

La liste des 3 commits est affichée avec l'heure, la date, l'auteur et le message





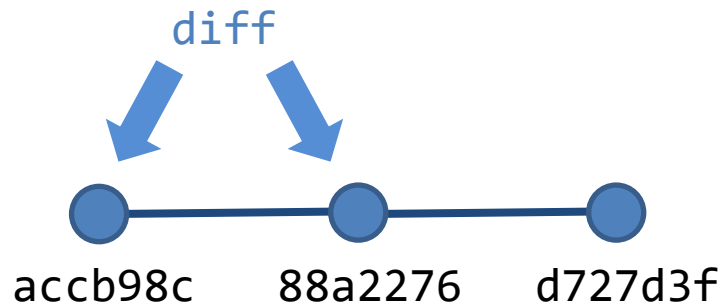
# git en local - les bases

## Comparer des commits

```
git diff accb98c 88a2276
diff --git a/solver.py b/solver.py
index 4bbf4e3..684d3de 100644
--- a/solver.py
+++ b/solver.py
@@ -1,3 +1,5 @@
+#!/usr/bin/env python
+
import math

print math.sin(1.0)
```

Différences au format  
"diff" (Linux).  
Les 2 lignes vertes ont  
été ajoutées.



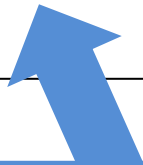


# git en local - les bases

On enrichit le code

```
vi README.md          # ajout d'un fichier README
mkdir tools            # crée un répertoire
cd tools
touch __init__.py      # crée un fichier (module python)
vi fct.py              # crée un module tools.fct
cd ..
vi solver.py           # modifie le code

python solver.py       # lance le code
```



l'exécution de `solver.py` va  
créer des fichiers `*.pyc`  
qu'on ne veut pas commiter!



# git en local - les bases

On "ajoute" tout de même...

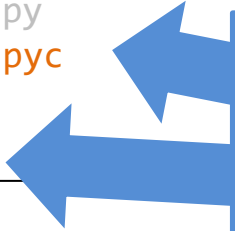
```
git add .          # '.' ajoute toutes les modifications
git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

```
new file:   README.md
modified:   solver.py
new file:   tools/__init__.py
new file:   tools/__init__.pyc
new file:   tools/fct.py
new file:   tools/fct.pyc
```



On ne veut pas  
commiter ces  
deux fichiers!



# git en local - les bases

## Supprimer un ajout à l'index

```
git reset HEAD          # annule le "git add"

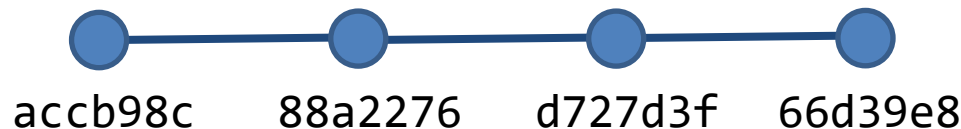
vi .gitignore
```

On crée un fichier  
avec extensions à  
ignorer

.gitignore

```
*.pyc
*~
```

```
git add .                # les fichiers *.pyc sont ignorés!
git commit -m "new code structure"
[master 66d39e8] new code structure
5 files changed, 15 insertions(+), 2 deletions(-)
...
```





# git en local - les bases

## Déplacer, renommer un fichier

```
git mv README.md README.txt
```

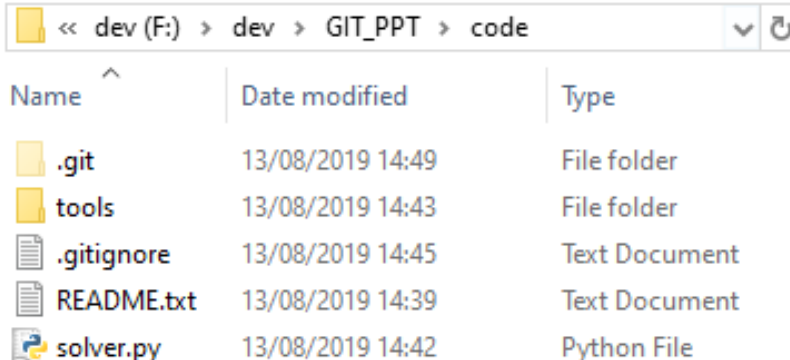
En pratique, git va détecter automatiquement un "rename" manuel. La commande précédente est équivalente à:

```
mv README.md README.txt
git add .
git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    README.md -> README.txt
```

Enfin, on commite:

```
git commit -m "change extension"
[master d557073] change extension
...
```



<< dev (F:) > dev > GIT_PPT > code			▼	↺
Name	Date modified	Type		
.git	13/08/2019 14:49	File folder		
tools	13/08/2019 14:43	File folder		
.gitignore	13/08/2019 14:45	Text Document		
README.txt	13/08/2019 14:39	Text Document		
solver.py	13/08/2019 14:42	Python File		



# git en local - les bases

## ***Supprimer un fichier***

```
git rm README.txt
```

équivalent à:

```
rm README.txt
git add README.txt          # yes: "add"!
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

       deleted:    README.txt
```

Si on s'est trompé, on récupère README.txt  
(en suivant pour l'instant aveuglément les instructions de `git status`)

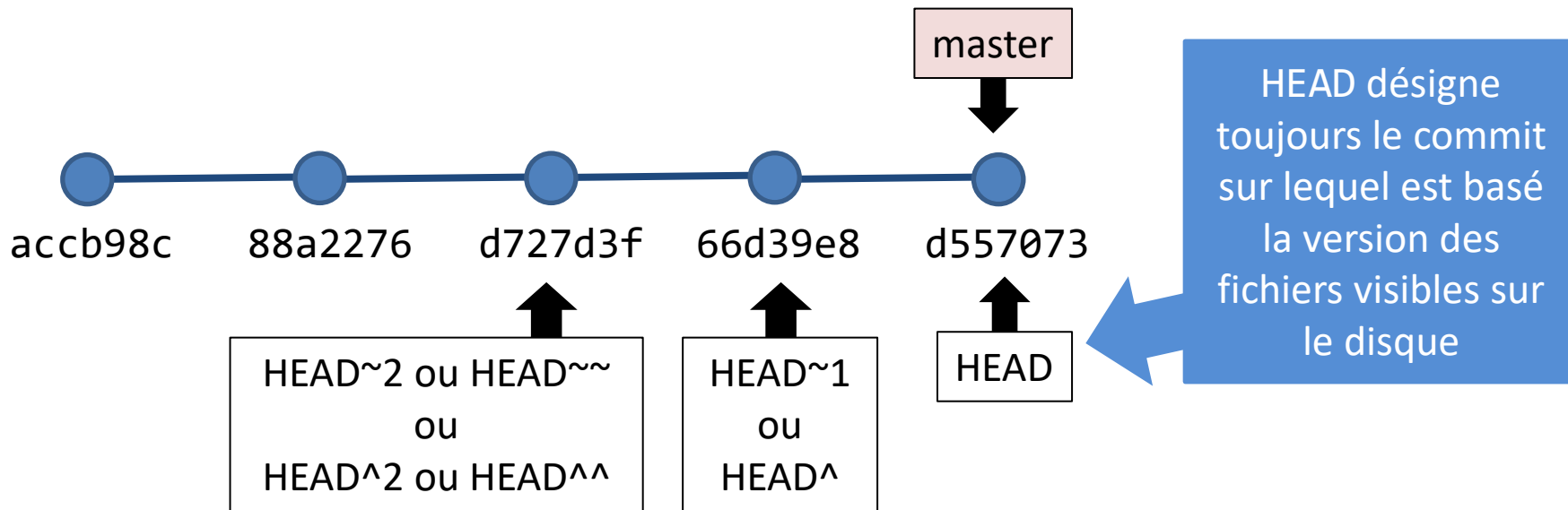
```
git reset HEAD README.txt    # supprime le "add"
git checkout -- README.txt    # récupère le fichier
```



# git en local - les bases

## Identifier des versions du code

En plus des numéros de commits, git utilise toutes sortes de **pointeurs** (imaginer un drapeau) qui facilitent la désignation d'un commit



Example: `git diff d727d3f 66d39e8`

équivalent à

`git diff HEAD~~ HEAD~` ou `git diff master~~ master~`

sous Windows, privilégier ~ à ^ car ^ est devenu un caractère d'échappement du terminal! autre solution: guillemets: "HEAD^"

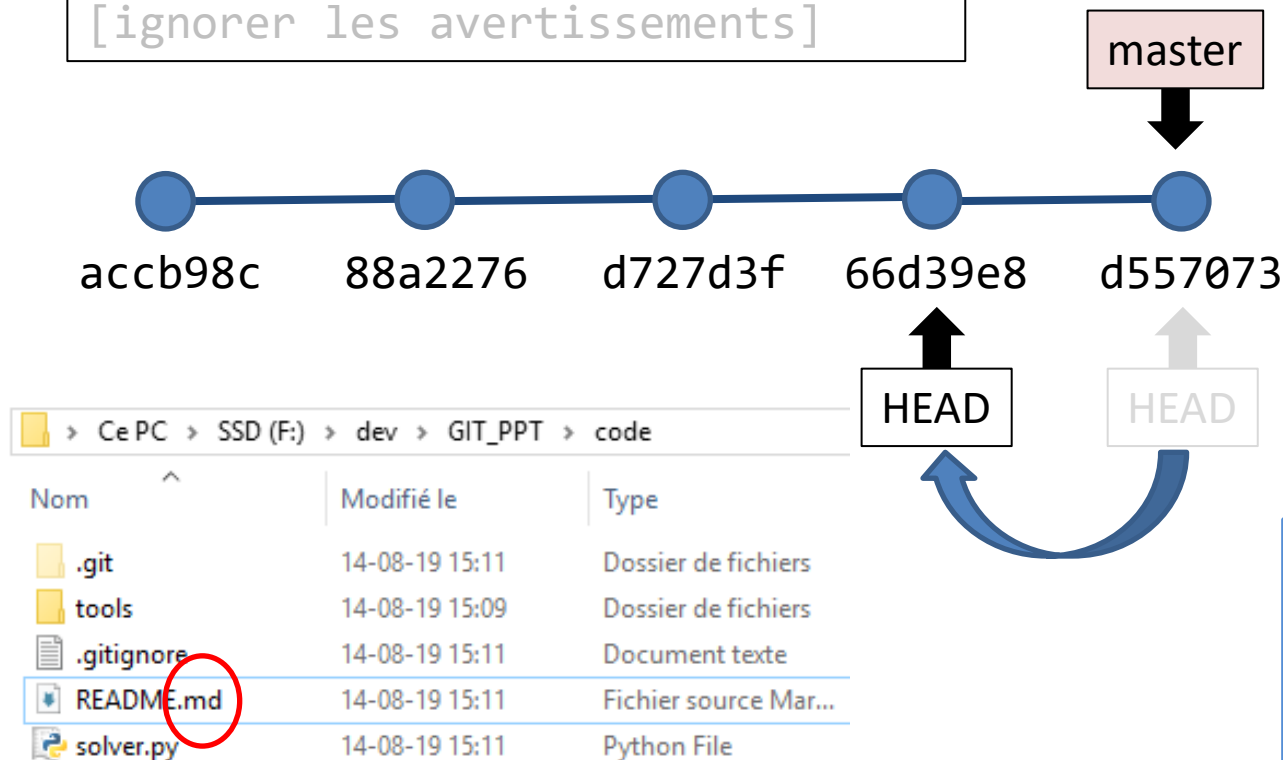


# git en local - les bases

## *Se déplacer dans l'arbre des versions*

Pour récupérer une version donnée, on utilise la commande **checkout**

```
git checkout 66d39e8  
[ignorer les avertissements]
```



checkout déplace le pointeur HEAD (et adapte les fichiers sur le disque)





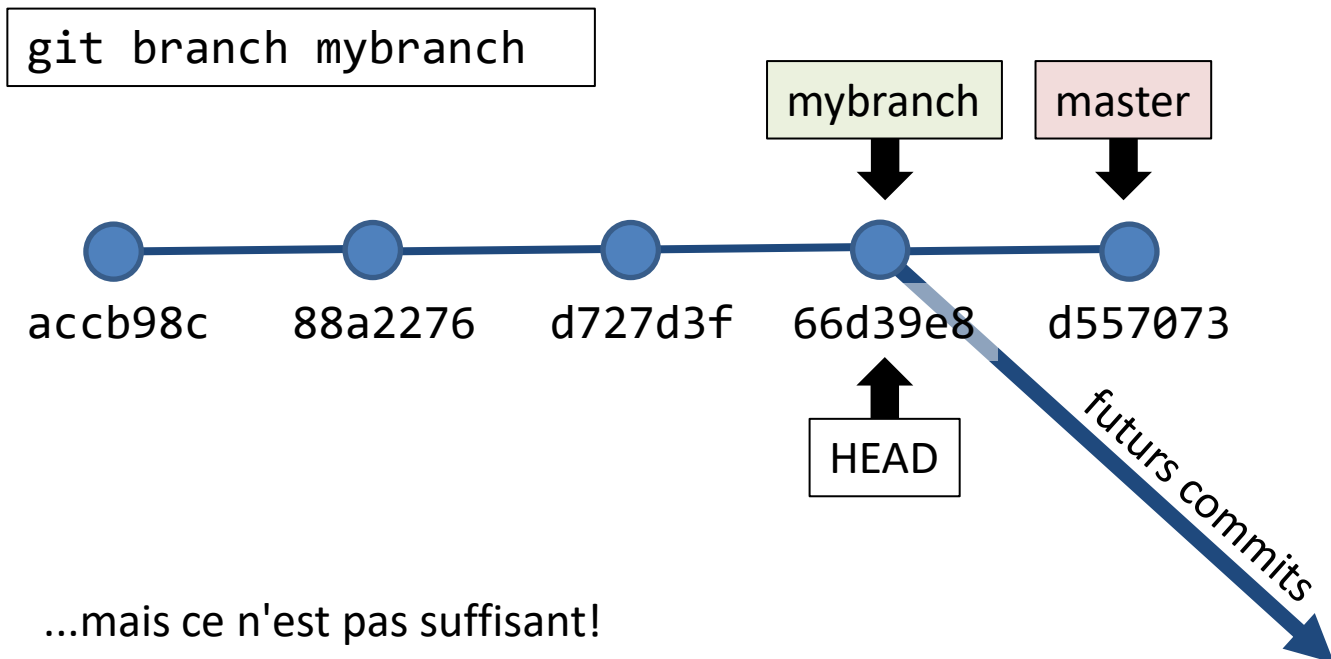
# git en local - les bases

## Créer une branche nommée "mybranch"

Il est tout à fait possible de modifier cette version et effectuer de nouveaux commits à partir de ce point.

Il faut alors créer une "**branche**" qui va **suivre nos commits**.

La branche est nommée par un **pointeur** tel "master" pour la branche initiale.



## A RETENIR

une branche  
=  
un pointeur

...mais ce n'est pas suffisant!



# git en local - les bases

## Travailler dans la nouvelle branche (1/4)

git status nous renseigne sur l'état de git:

cette zone indique un numéro de commit et non une branche

```
r_bom@KRABS MINGW64 /f/dev/GIT_PPT/code ((66d39e8...))  
$ git status  
HEAD detached at 66d39e8  
nothing to commit, working tree clean
```

git attire notre attention sur le fait que nous sommes "détaché" de toute branche.

La branche a bien été créée mais nous ne travaillons pas dessus!



# git en local - les bases

## Travailler dans la nouvelle branche (2/4)

```
git checkout mybranch
```

le pointeur "mybranch" va suivre  
nos futurs commits!

```
$ git checkout mybranch
Switched to branch 'mybranch'

r_bom@KRABS MINGW64 /f/dev/GIT_PPT/code (mybranch)
$ |
```

### Remarque:

Les 2 commandes précédentes  
(branch + checkout) peuvent être  
fusionnées en 1 seule:

```
git checkout -b mybranch
```

## A RETENIR

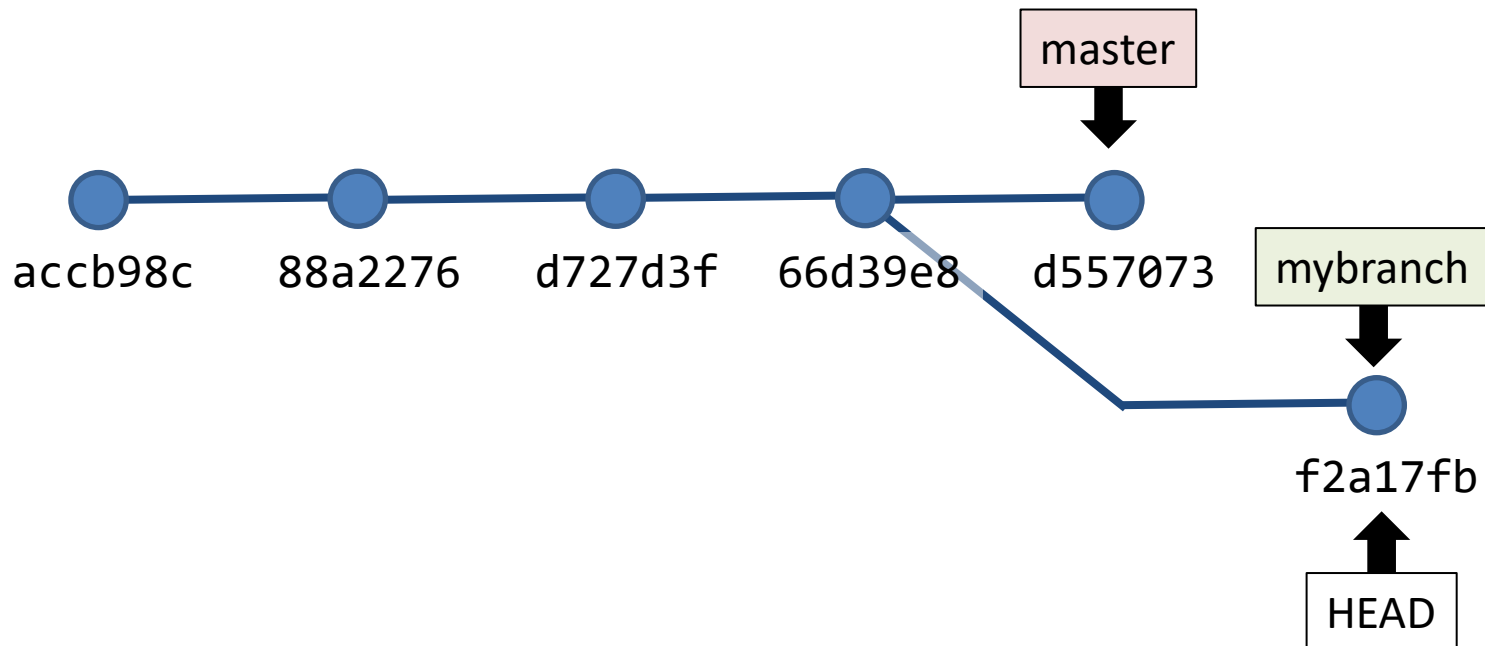
une branche  
=  
un pointeur  
...qui suit notre travail  
lors de certaines commandes git  
(commit, merge, etc.)



# git en local - les bases

## Travailler dans la nouvelle branche (3/4)

```
vi solver.py          # modification du fichier solver.py
git add solver.py
git commit -m "better input"
[mybranch f2a17fb] better input
1 file changed, 1 insertion(+), 1 deletion(-)
```





# git en local - les bases

## Travailler dans la nouvelle branche (4/4)

Avec git, l'arbre des révisions peut devenir très compliqué.  
Il est dès lors très important de bien connaître à tout instant l'état du repository, l'état des différentes branches et l'endroit où on se trouve (HEAD)

```
git branch
```

```
  master  
* mybranch
```

Liste les branches  
la branche avec une étoile et la  
branche active

```
git diff master  
[...]
```

Affiche les différences entre la  
branche courante et "master"

```
git log --graph --oneline --all
```

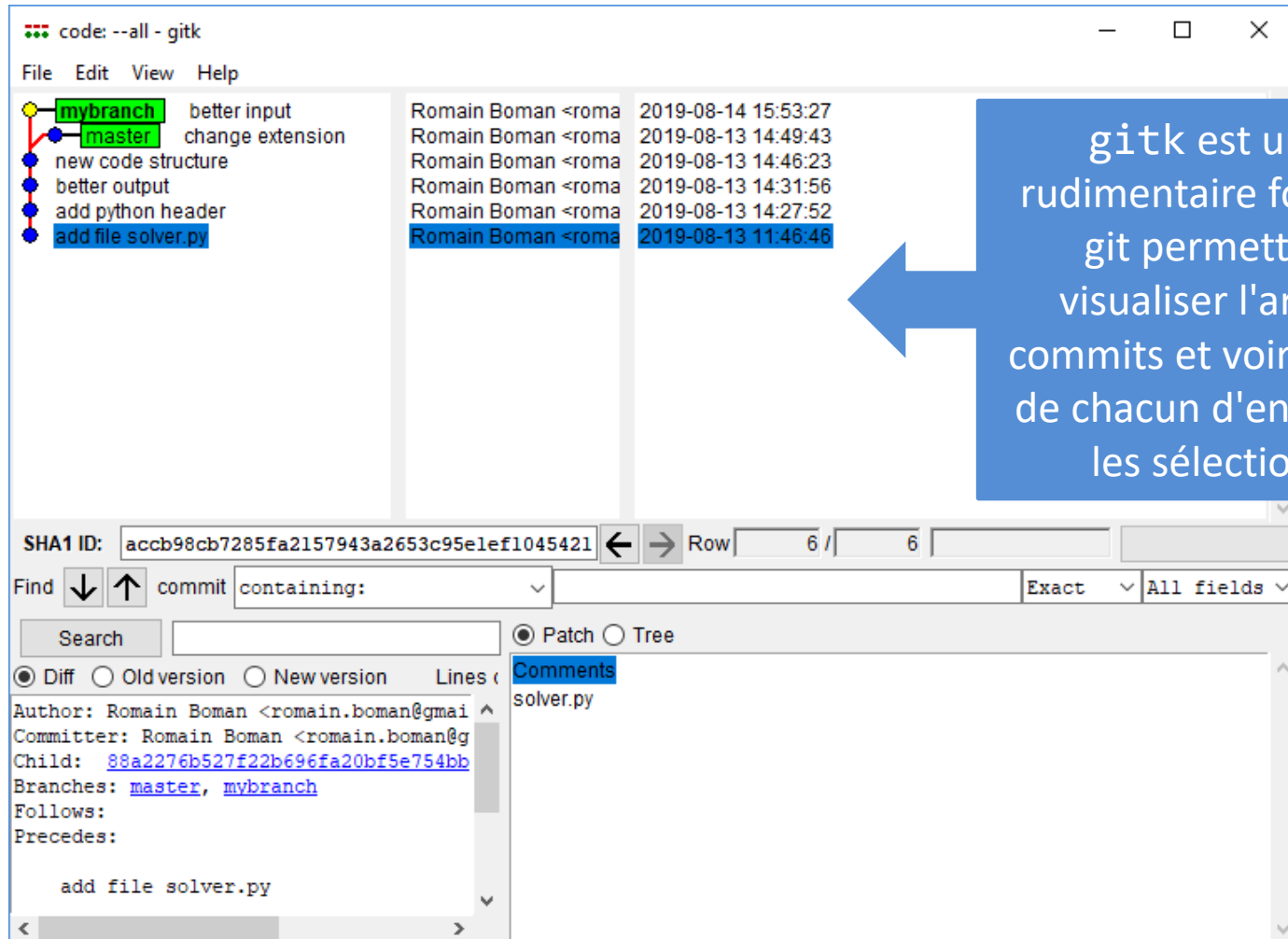
```
* f2a17fb (HEAD -> mybranch) better input  
| * d557073 (master) change extension  
|/  
* 66d39e8 new code structure  
* d727d3f better output  
* 88a2276 add python header  
* accb98c add file solver.py
```

Affiche une  
représentation  
graphique de l'arbre



# git en local - les bases

```
gitk --all
```



gitk est un outil rudimentaire fourni avec git permettant de visualiser l'arbre des commits et voir les détails de chacun d'entre eux en les sélectionnant

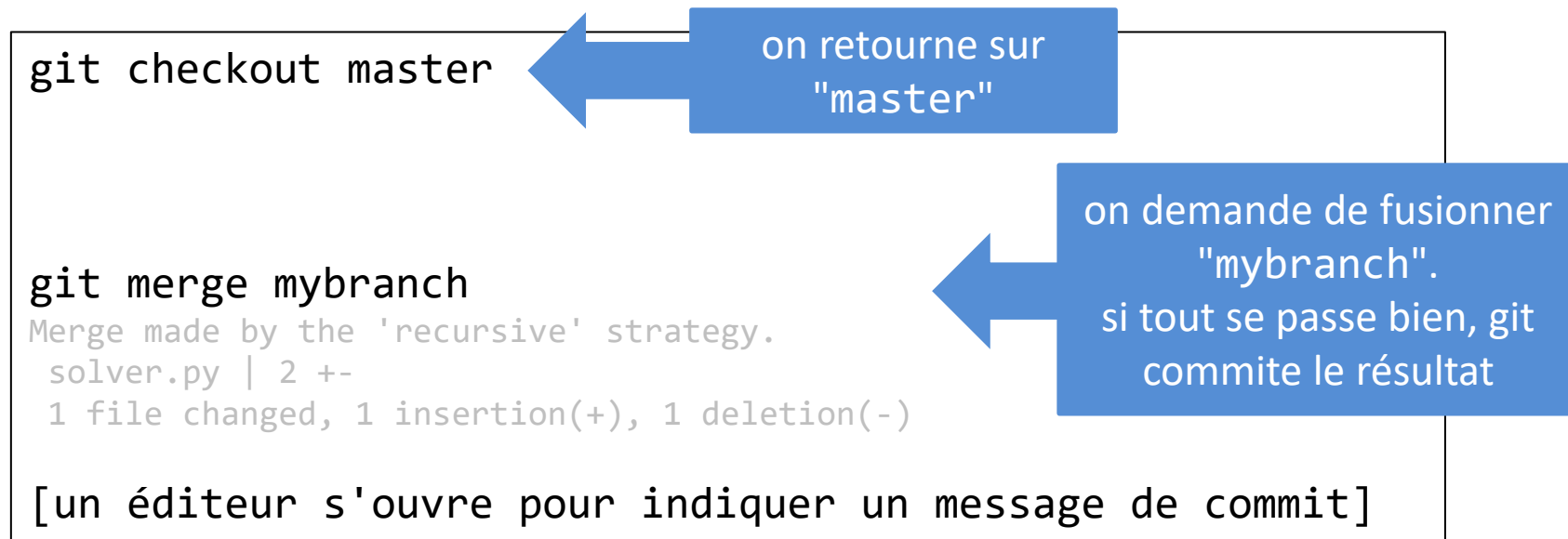


# git en local - les bases

## *Fusionner des branches*

La puissance de git par rapport à d'autres SCM est sa capacité à fusionner simplement des branches.

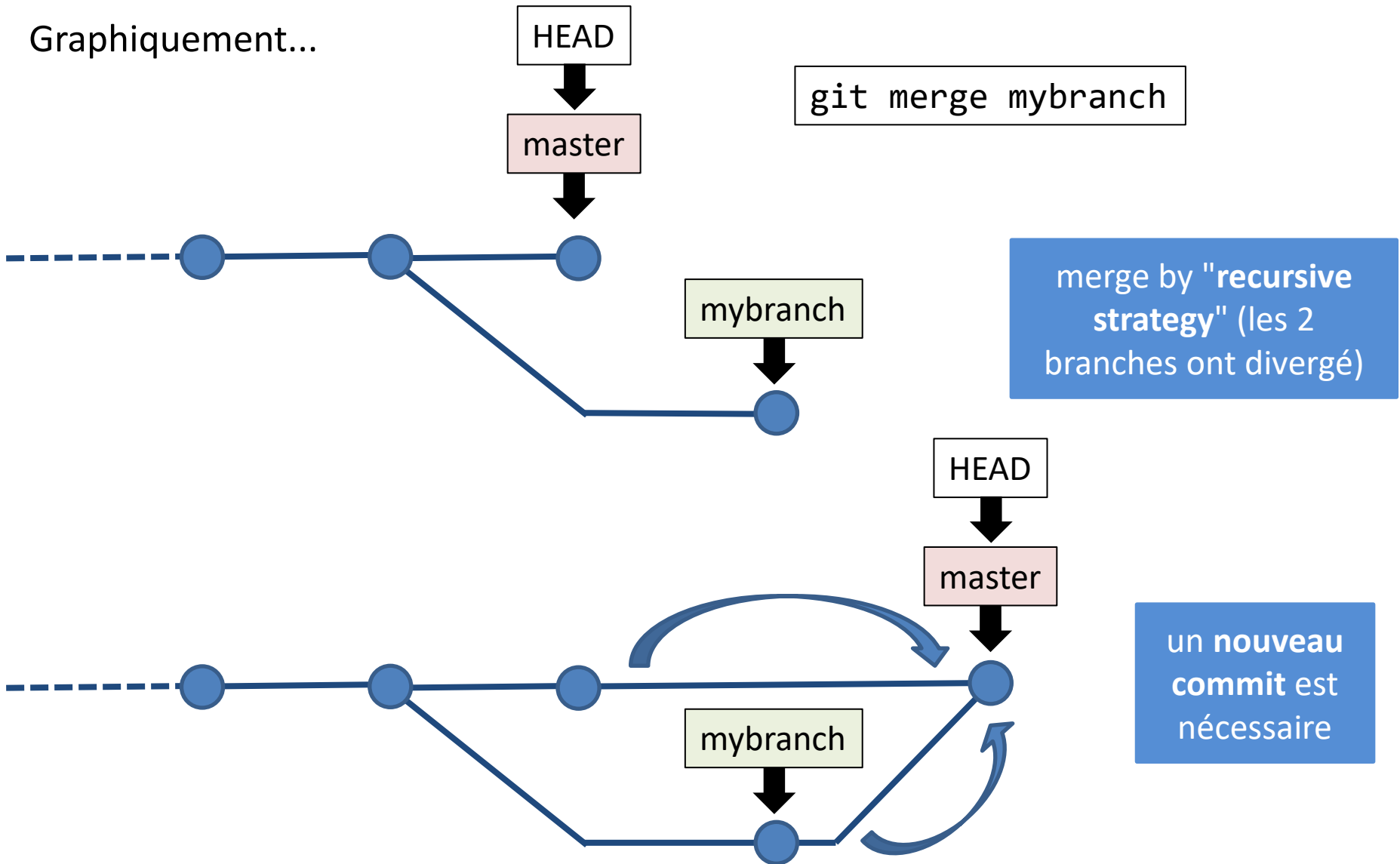
Imaginons que l'on veuille fusionner les modifications développées de la branche "mybranch" à la branche "master".





# git en local - les bases

Graphiquement...







# git en local - les bases

*Comment mettre à jour "mybranch" sur "master"?*

```
git checkout mybranch
```

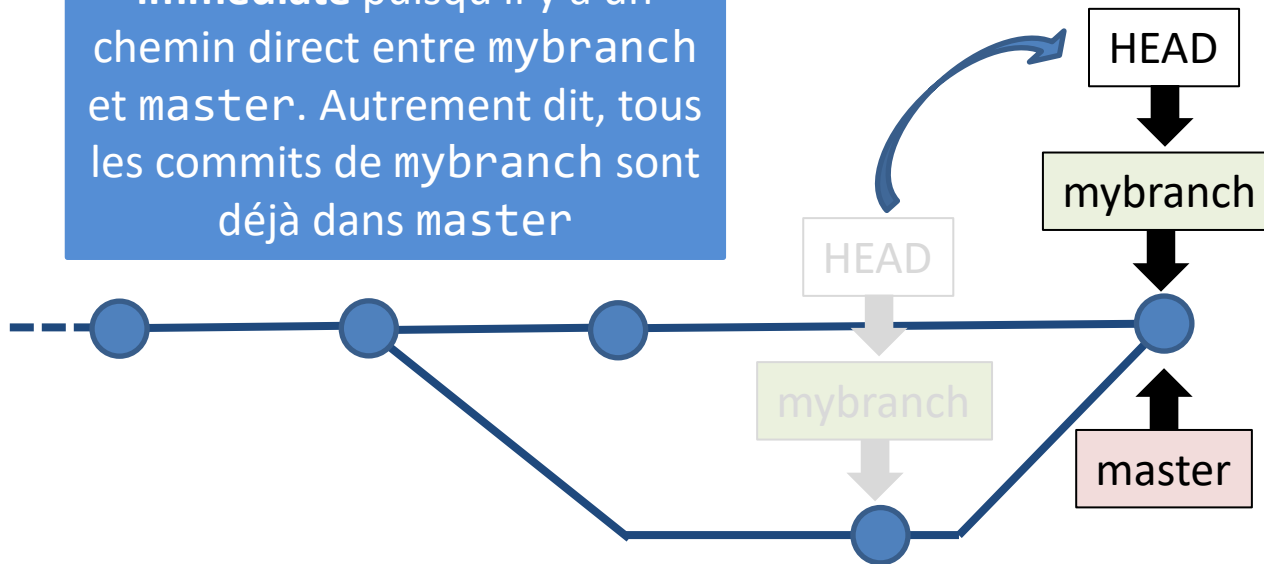
```
git merge master
```

```
Updating f2a17fb..935530d
```

```
Fast-forward
```

On retourne sur  
"mybranch"

L'opération de merge ici est **immédiate** puisqu'il y a un chemin direct entre mybranch et master. Autrement dit, tous les commits de mybranch sont déjà dans master



On parle ici de **"fast-forward merge"** qui consiste simplement à déplacer le pointeur de branche sans faire de nouveau commit



# git en local - les bases

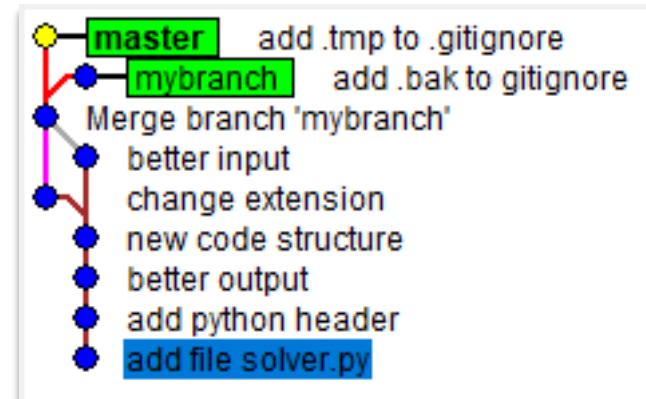
## *Que se passe-t-il si un "recursive merge" se passe mal? (1/4)*

Modifions 2x le même fichier: une première fois dans mybranch et une seconde fois dans master

```
git checkout mybranch
vi .gitignore
[ajouter une extension à ignorer]
git add .gitignore
git commit -m "add .bak to .gitignore"

git checkout master
vi .gitignore
[ajouter une extension à ignorer]
git add .gitignore
git commit -m "add .tmp to .gitignore"

gitk --all
```





# git en local - les bases

## *Que se passe-t-il si un "recursive merge" se passe mal? (2/4)*

Fusionnons mybranch dans master...

```
git merge mybranch
```

```
Auto-merging .gitignore
```

```
CONFLICT (content): Merge conflict in .gitignore
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

```
git status
```

```
On branch master
```

```
You have unmerged paths.
```

```
(fix conflicts and run "git commit")
```

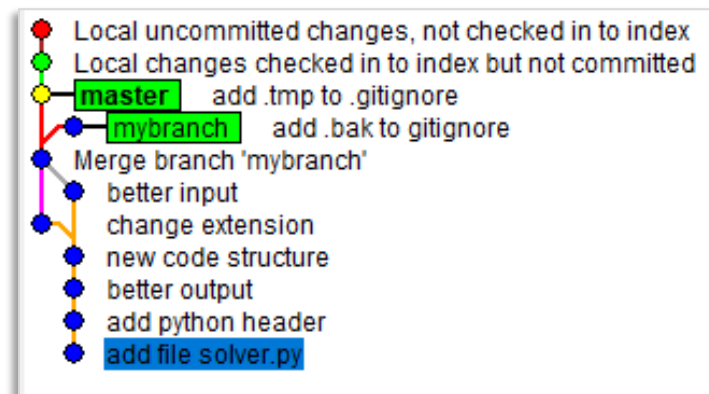
```
(use "git merge --abort" to abort the merge)
```

```
Unmerged paths:
```

```
(use "git add <file>..." to mark resolution)
```

```
both modified: .gitignore
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```





# git en local - les bases

## Que se passe-t-il si un "recursive merge" se passe mal? (3/4)

Résolution manuelle des conflits...

```
vi .gitignore
```

```
*.pyc
*~
<<<<<< HEAD
*.tmp
=====
*.bak
>>>>>> mybranch
```

Le conflit est entouré des symboles  
<<<<<< et >>>>>>

Résoudre le conflit consiste à  
supprimer (intelligemment) cette  
zone pour fusionner les  
développements

Au dessus: le code venant de la  
branche actuelle (HEAD), c'est-à-  
dire master.

En dessous: le code de mybranch.

Dans ce cas-ci, on  
conserve les 2  
lignes ajoutées

```
*.pyc
*~
*.tmp
*.bak
```



# git en local - les bases

## *Que se passe-t-il si un "recursive merge" se passe mal? (4/4)*

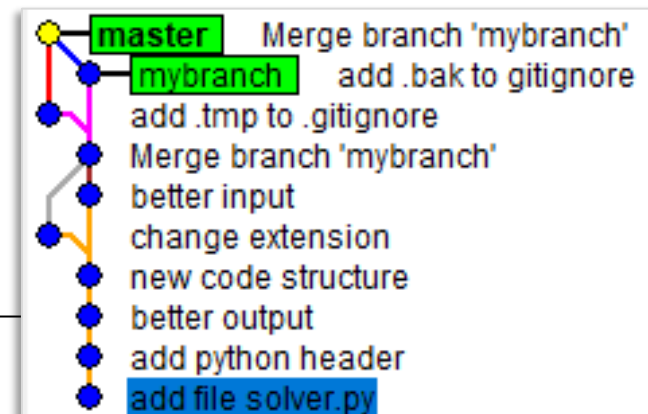
A ce stade, git a placé dans l'index ("staging area") toutes les fusions de fichiers qui ont été effectuées avec succès (dans notre cas, il n'y en a pas)

Il suffit donc d'ajouter le fichier qui a été fusionné manuellement

```
git add .gitignore
```

et ensuite créer le commit du merge:

```
git commit  
  
[un éditeur s'ouvre - adapter  
le message si nécessaire]  
  
gitk --all
```





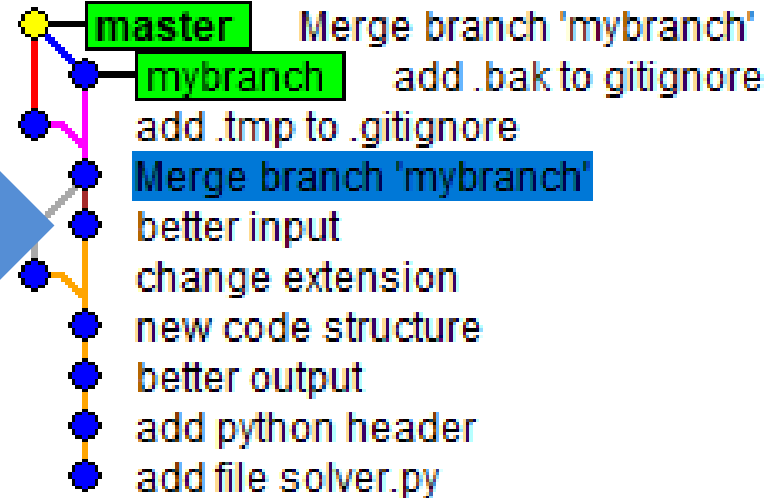
# git en local - les bases

## Remarque importante (1/3)

Pratiquement, qu'appelle-t-on "branche"?

Ce commit particulier, créé dans la branche "mybranch" fait maintenant partie des 2 branches ("mybranch" mais aussi "master")

Une branche rassemble tous les commits en dessous de lui, ce n'est donc certainement pas une "ligne de commits", mais plutôt **tous les commits dont dépend le commit pointé par le pointeur de branche.**



En particulier, le commit initial appartient toujours à toutes les branches!



# git en local - les bases

## ***Remarque importante (2/3):***

Conséquence: l'idée de faire une branche contenant toutes les versions "stables" d'un code n'est pas possible.

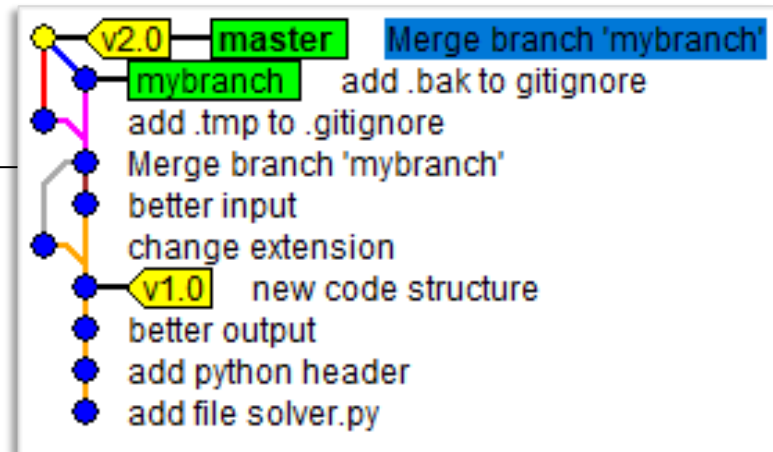
Après un merge, il sera impossible de distinguer une version stable d'un commit précédent provenant d'une branche de développement!



# git en local - les bases

## Remarque importante (3/3):

Solution: **les tags** - Un tag est une étiquette sur un commit.



```
git checkout 66d39e8
git tag v1.0
git checkout master
git tag v2.0
```

```
git tag
```

```
v1.0
```

```
v2.0
```

Liste les tags

```
git checkout v1.0
git diff v1.0 v2.0
```

Les tags permettent de retrouver facilement les versions en oubliant leurs numéros de commit





# git en local - les bases

## *Tag vs branche?*

Une branche et un tag sont très similaires: ce sont des "étiquettes" liées à un commit particulier.

La seule différence est le fait qu'**un tag est fixe** contrairement à une branche qui "se déplace" suite à certaines commandes git (`git commit`, `git merge`, etc.)

## *Supprimer un tag ou une branche*

On utilise simplement la même commande que pour la création avec l'option `-d`

```
git branch -d mybranch  
Deleted branch mybranch (was 620b892).
```

```
git tag -d v2.0  
Deleted tag 'v2.0' (was a227c19)
```

# Utiliser git

PARTIE 1  
Utiliser git en local

Je nettoie mon travail  
avant de le partager



# git en local - nettoyage

## **Nettoyer son répertoire de travail**

(Supprimer les fichiers non connus par git)

```
git status
```

```
On branch master
```

```
Untracked files:
```

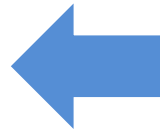
```
(use "git add <file>..." to include in what will be committed)
```

```
results/  
tmp.txt
```

```
git clean -fd
```

```
Removing results/
```

```
Removing tmp.txt
```



Supprime tous les fichiers et  
répertoires non gérés par git

(sauf ceux du .gitignore -  
ajouter -x pour supprimer les  
fichiers du .gitignore)

Les fichiers sont définitivement perdus (pas de corbeille!)

Conseil: faire un backup avant `git clean`



# git en local - nettoyage

## **Récupérer son répertoire de travail dans l'état initial**

(Supprimer les modifications de son répertoire de travail)

```
git status
```

```
On branch master
```

```
Changes not staged for commit:
```

```
(use "git add/rm <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified:   README.txt
deleted:    tools/__init__.py
deleted:    tools/fct.py
```

```
git reset --hard HEAD
```

```
HEAD is now at a227c19 Merge branch 'mybranch'
```

rétablit tous les fichiers gérés par git  
dans leur état HEAD

Conserve tous les fichiers non gérés  
par git  
(voir `git clean` pour les  
supprimer)

Ici les modifications de `README.txt` qui  
n'ont jamais été commitées ont été  
définitivement perdues!

Conseil: faire un backup avant `git reset`



# git en local - nettoyage

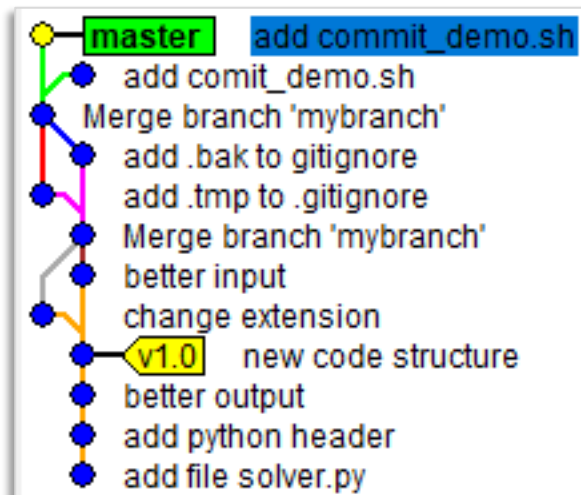
## Modifier le message du dernier commit

```
vi commit_demo.sh
git add commit_demo.sh
git commit -m "add comit_demo.sh"
[master 332f55d] add comit_demo.sh
```

on crée un nouveau fichier mais on se trompe dans le message de commit

```
git commit --amend -m "add commit_demo.sh"
[master be57369] add commit_demo.sh
```

commit --amend permet de corriger



commit --amend n'a pas modifié l'ancien commit mais a créé un nouveau!

Il est important de comprendre que git ne modifie **JAMAIS** un ancien commit

Le vieux commit inutile sera supprimé plus tard lors de l'exécution du "**garbage collector**" (git gc)



# git en local - nettoyage

## Modifier/Amender le contenu du dernier commit (1/3)

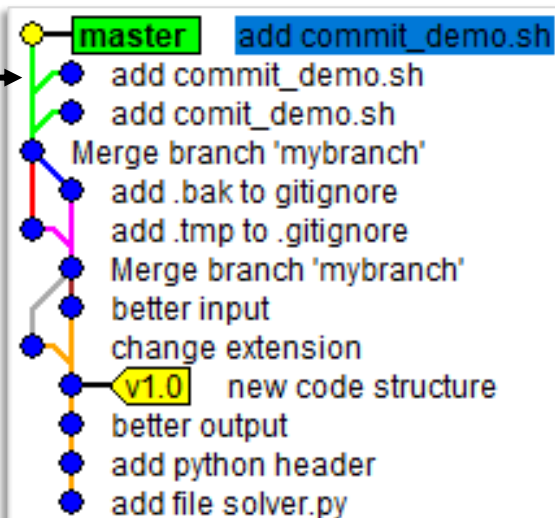
`git commit --amend` permet de modifier et d'enrichir le dernier commit bien au delà du message de commit. Ci-dessous, on corrige un bug qui a été commité.

```
vi commit_demo.sh  
git add commit_demo.sh
```

on édite le script et on procède  
comme un commit classique

```
git commit --amend -m "add commit_demo.sh"  
[master 0df2d7e] add commit_demo.sh
```

be57369



A nouveau, rien a été effacé. Il  
suffirait d'exécuter

```
git reset --hard be57369
```

pour déplacer master sur le  
commit précédent et récupérer  
l'état initial



# git en local - nettoyage

## ***Modifier/Amender le contenu du dernier commit (2/3)***

`git commit --amend` crée un commit parallèle au dernier commit en "fusionnant" le dernier commit et les nouvelles modifications mise dans l'index (staging area).

Dans ce contexte, une commande utile est `git reset` couplée à un nom de fichier. Elle permet de récupérer un fichier dans une version particulière (par exemple la version avant le commit qu'on veut amender!

```
git reset HEAD~1 file.txt
```

La version précédente est placée dans la staging area,  
La version sur disque n'est pas modifiée

```
git commit --amend -m "commit sans modifier file.txt"
```

Très utile pour supprimer des mods qu'on aurait commitées par erreur!



# git en local - nettoyage

## *Modifier/Amender le contenu du dernier commit (3/3)*

Imaginons un cas plus complexe où le dernier commit est riche (beaucoup de modifications) et on aimerait **revenir au moment juste avant de faire le commit précédent** pour corriger certaines choses.

Il est nécessaire de mieux comprendre la commande `git reset`

`git reset` permet de contrôler l'état des **3 "versions"** du code qu'on manipule avec les commandes `git`:

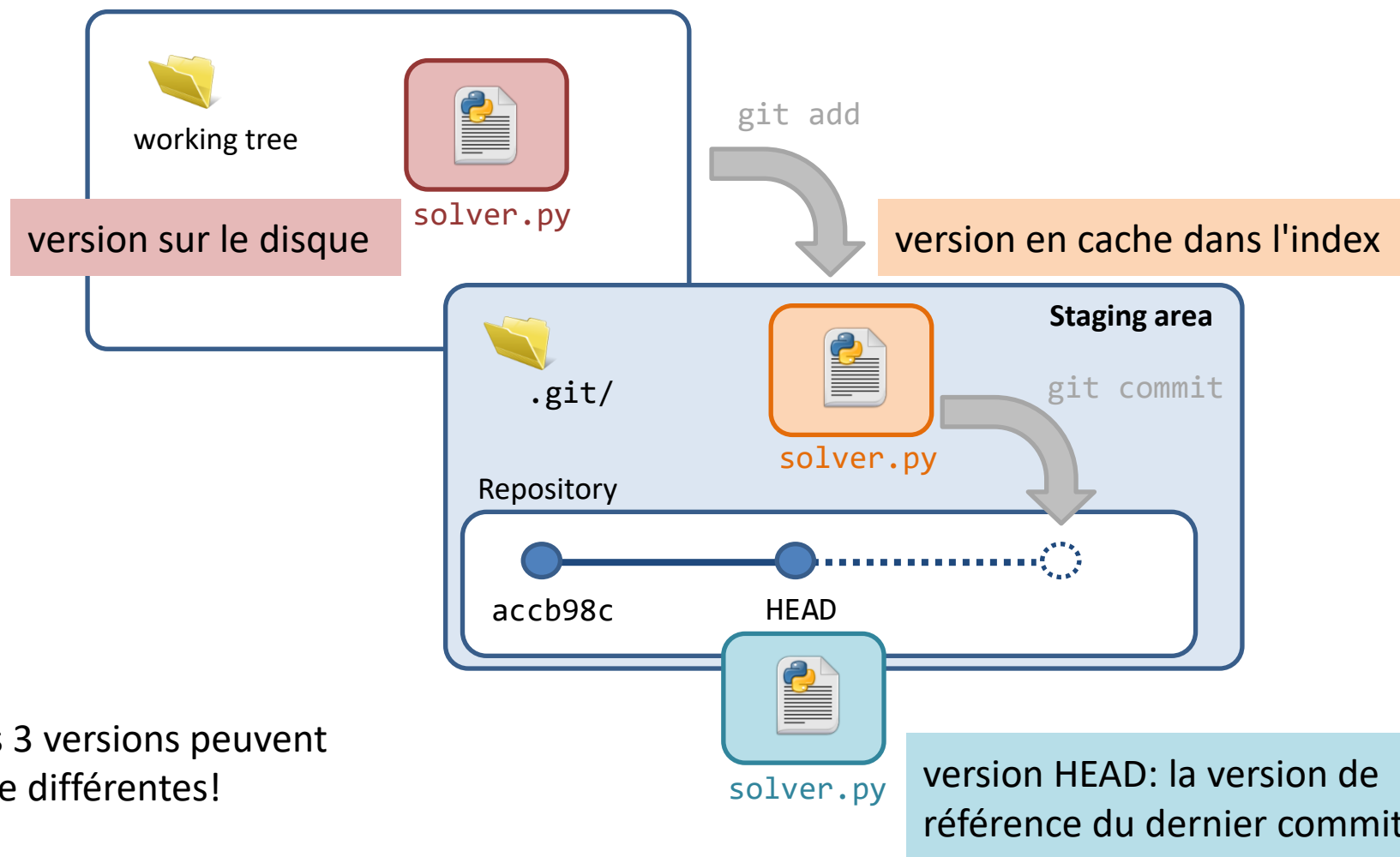
- **version 1:** "HEAD": le pointeur vers le commit de référence du repository; la branche actuelle (master p expl) suit ce pointeur s'il change,
- **version 2:** la "staging area": il s'agit de HEAD + les modifications qu'on a ajoutées avec `git add`, en attente d'un commit,
- **version 3:** la "copie de travail", c'est-à-dire les fichiers sur le disque.





# git en local - nettoyage

RAPPEL: Mécanisme en 2 temps, pas très intuitif:





# git en local - nettoyage

## Mieux comprendre git reset (1/2)

git reset peut agir sur les 3 versions:

	HEAD	staging area	working copy
--soft	changée	<i>inchangée</i>	<i>inchangée</i>
--mixed (default)	changée	changée	<i>inchangée</i>
--hard	changée	changée	changée

git reset --soft HEAD~

- recule HEAD d'un commit. Les fichiers sur disque et la staging area restent dans leur état actuel.
- Si notre copie sur disque était propre et correspondait à HEAD, on se retrouve dans l'état après le git add et **avant** le git commit précédent.



# git en local - nettoyage

## Mieux comprendre git reset (2/2)

git reset peut agir sur les 3 versions:

	HEAD	staging area	working copy
--soft	changée	<i>inchangée</i>	<i>inchangée</i>
--mixed (default)	changée	changée	<i>inchangée</i>
--hard	changée	changée	changée

git reset --mixed HEAD~                      ou                      git reset HEAD~

- recule HEAD et la staging area d'un commit. Les fichiers sur disque ne sont pas touchés.
- si notre copie sur disque était propre et correspondait à HEAD, on se retrouve dans l'état avant le git add et **avant** le git commit précédent.



# git en local - nettoyage

## *Jouer un commit "à l'envers": git revert*

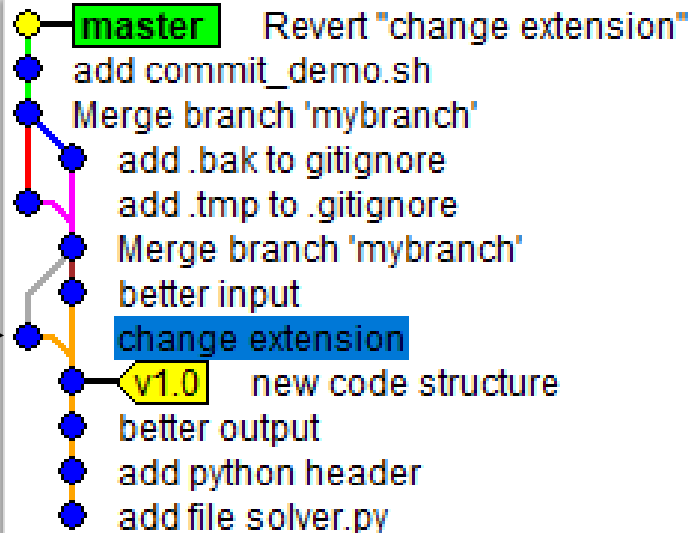
Pour comprendre cette commande, chaque commit doit être vu comme une **modification de ce qui précède** et non comme un état du code.

Ces modifications peuvent être appliquées "**à l'envers**" pour annuler un commit précédent

```
git revert d557073
```

```
[master f7495f4] Revert "change extension"  
1 file changed, 0 insertions(+), 0 deletions(-)  
rename README.txt => README.md (100%)
```

On applique le commit  
d557073 à l'envers



Un nouveau commit est  
créé qui change juste  
l'extension du fichier.

Le contenu du fichier  
n'est pas impacté.

Il ne s'agit donc pas de  
retrouver l'état du code  
au commit d557073

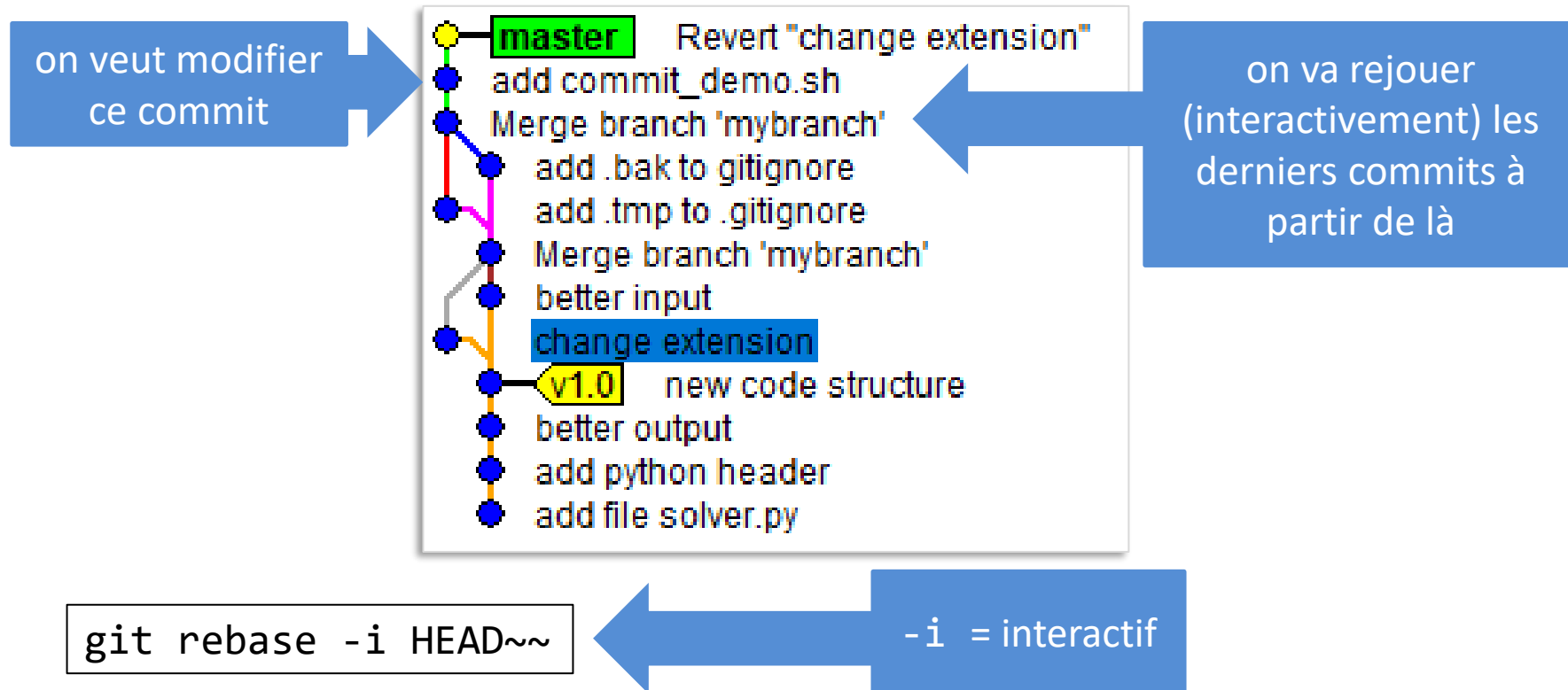


# git en local - nettoyage

## ***Modifier l'historique plus en amont: le "Rebasing"***

Imaginons que nous voulions corriger un bug dans l'avant-dernier commit

L'idée est de rejouer les 2 derniers commits (HEAD et HEAD~) à partir du commit qui précède (HEAD~~)





# git en local - nettoyage

```
git rebase -i HEAD~~
```

MINGW64:/f/dev/GIT\_PPT/code

```
pick 0df2d7e add commit_demo.sh
pick f7495f4 Revert "change extension"
```

```
# Rebase a227c19..f7495f4 onto a227c19 (2 commands)
```

```
#
```

```
# Commands:
```

```
# p, pick <commit> = use commit
```

```
# r, reword <commit> = use commit, but edit the commit message
```

```
# e, edit <commit> = use commit, but stop for amending
```

```
# s, squash <commit> = use commit, but meld into previous commit
```

```
# f, fixup <commit> = like "squash", but discard this commit's
```

```
# x, exec <command> = run command (the rest of the line) using
```

```
# b, break = stop here (continue rebase later with 'git rebase
```

```
# d, drop <commit> = remove commit
```

```
# l, label <label> = label current HEAD with a name
```

```
# t, reset <label> = reset HEAD to a label
```

```
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
```

```
# . create a merge commit using the original merge commit
```

```
# . message (or the oneline, if no original merge commit
```

```
# . specified). Use -c <commit> to reword the commit message
```

```
#
```

```
# These lines can be re-ordered; they are executed from top to
```

```
#
```

```
# If you remove a line here THAT COMMIT WILL BE LOST.
```

```
#
```

```
<_PPT/code/.git/rebase-merge/git-rebase-todo [unix] (17:00 15/08/2019)1,1 Haut
```

Un éditeur s'ouvre où une série de commandes sont pré-écrites.

Par défaut, `git rebase` va "pick" chaque commit successivement c'est-à-dire appliquer le 1er, puis le 2nd.


Ces lignes peuvent être modifiées (change l'opération), interverties (change l'ordre des commits), supprimées (supprime un commit), etc.

La sortie de l'éditeur démarrera le processus.



# git en local - nettoyage

On change la première commande en "edit" pour permettre l'édition



```
MINGW64:/f/dev/GIT_PPT/code
edit 0df2d7e add commit_demo.sh
pick f7495f4 Revert "change extension"
```

On sort de l'éditeur

```
r_bom@KRABS MINGW64 /f/dev/GIT_PPT/code (master)
$ git rebase -i HEAD^^
Stopped at 0df2d7e... add commit_demo.sh
You can amend the commit now, with


    git commit --amend

Once you are satisfied with your changes, run

    git rebase --continue

r_bom@KRABS MINGW64 /f/dev/GIT_PPT/code (master|REBASE-i 1/2)
$ |
```

La ligne de commande  
indique qu'on est en train  
d'effectuer un "rebase"



Le 1er commit a été appliqué sur HEAD~~.

On peut l'amender avec `git commit --amend`



# git en local - nettoyage

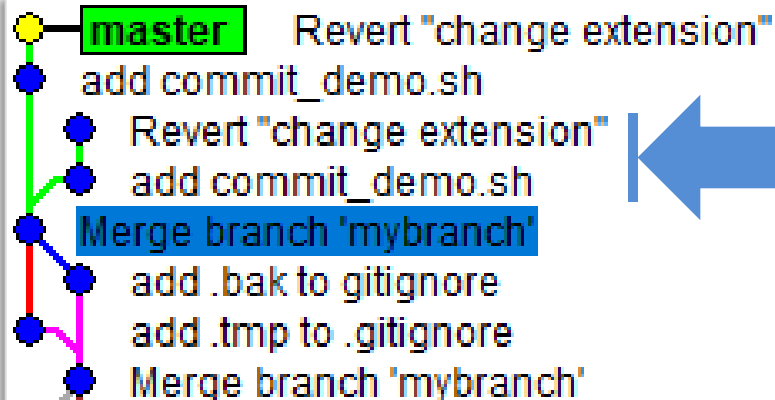
```
vi commit_demo.sh
git add commit_demo.sh
git commit --amend
[detached HEAD 6164d9a] add commit_demo.sh
Date: Thu Aug 15 13:59:03 2019 +0200
1 file changed, 10 insertions(+)
create mode 100644 commit_demo.sh

git rebase --continue
Successfully rebased and updated refs/heads/master.
```

← Edition de l'avant-dernier commit

← On termine le rebase (le dernier commit va être "pick" sur ce qu'on vient de faire)

Nouveaux commits (nouveaux numéros)



← Les anciens commits ne sont pas supprimés. Ils sont dans une branche qui n'est pas nommée.





# git en local - nettoyage

## ***Comment faire une édition fine du commit?***

Solution: combiner "git rebase" et "git reset"

```
git rebase -i HEAD~
```

```
[changer pick en edit]
```

```
git reset --soft HEAD~
```

On est dans l'état juste avant le git add / git commit problématique

Sans --soft, le git add serait déjà fait

```
[modifier le commit problématique]
```

```
[git add ...]
```

```
git commit -m "commit corrigé"
```

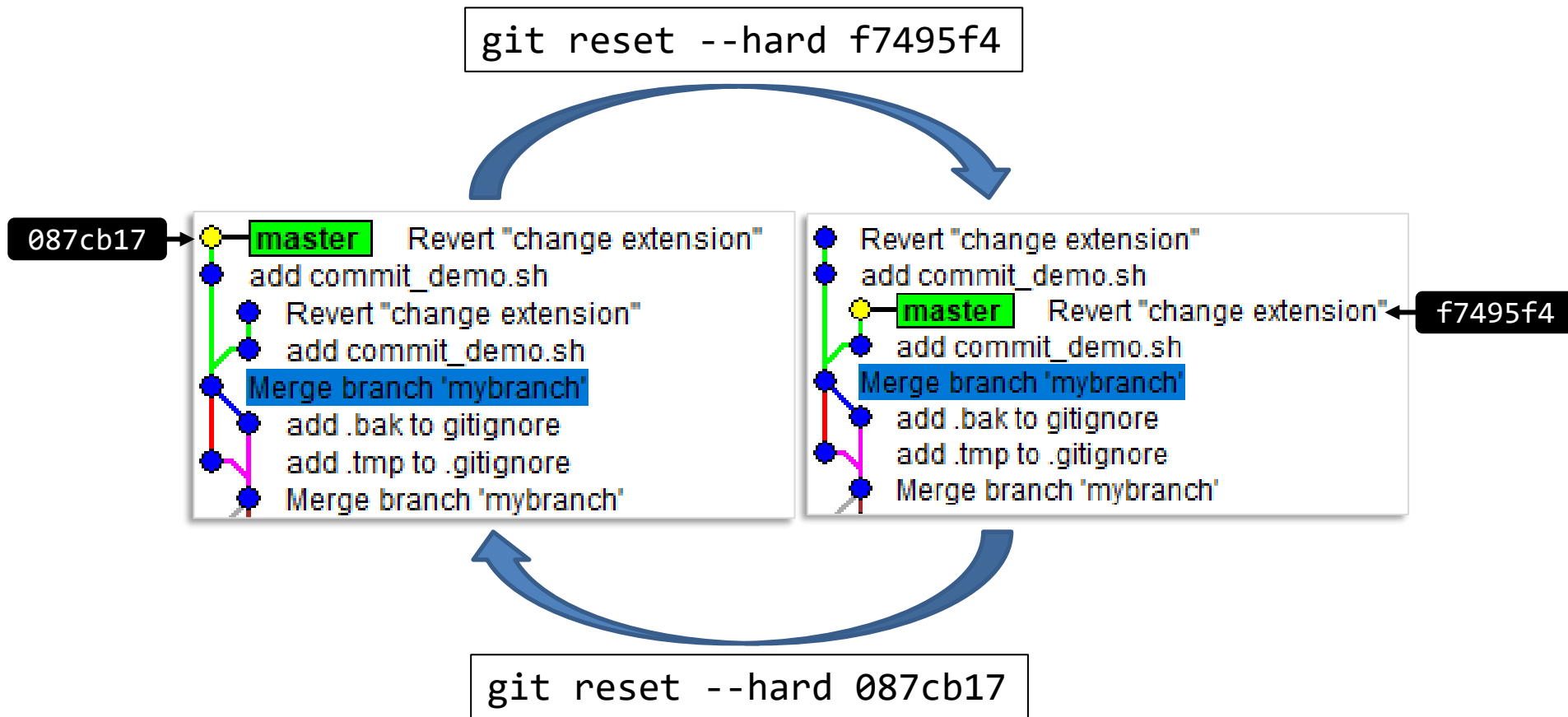
pas --amend ici!  
(on a reculé HEAD avec git reset)

```
git rebase --continue
```



# git en local - nettoyage

*Le rebase s'est mal passé?* ... pas de soucis!





# git en local - nettoyage

**Comment retrouver des numéros de commits dans des branches mortes?**  
(on parle de "lost commits" ou "dangling commits")

git garde une trace de tous les changements de HEAD.

```
git reflog
```

MINGW64:/f/dev/GIT\_PPT/code

```
087cb17 (HEAD -> master) HEAD@{0}: reset: moving to 087cb17
f7495f4 HEAD@{1}: reset: moving to f7495f4
087cb17 (HEAD -> master) HEAD@{2}: reset: moving to ORIG_HEAD
0df2d7e HEAD@{3}: reset: moving to ORIG_HEAD
087cb17 (HEAD -> master) HEAD@{4}: rebase -i (finish) returning to refs/heads/master
087cb17 (HEAD -> master) HEAD@{5}: rebase -i (pick): Revert "change extension"
6164d9a HEAD@{6}: commit (amend): add commit_demo.sh
0df2d7e HEAD@{7}: rebase -i: fast forward
a227c19 HEAD@{8}: rebase -i (start): checkout HEAD^
f7495f4 HEAD@{9}: checkout: moving from mybranch to master
f7495f4 HEAD@{10}: reset: moving to master
828e22b HEAD@{11}: commit (amend): Revert "change extension"
f7495f4 HEAD@{12}: reset: moving to HEAD
```

Conseil: Par précaution, faire un tag avant le rebase!

```
git tag avant_rebase
git rebase ...
```

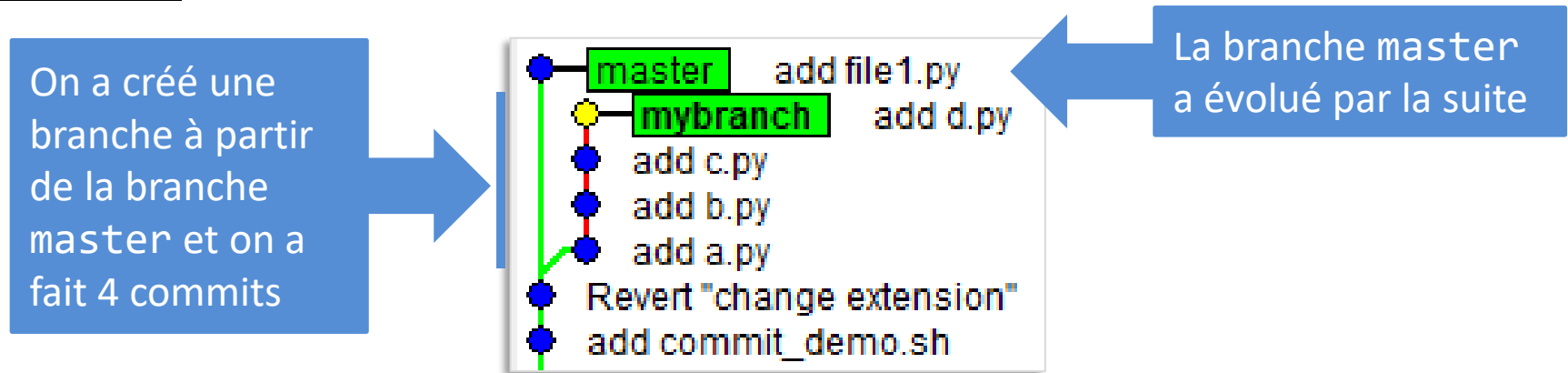


# git en local - nettoyage

`git rebase` peut également être utilisé pour **déplacer une branche**

C'est une autre manière de faire un merge (qui ne doit être utilisée qu'en local)

Etat initial:



On veut rejouer tous les commits qui ne sont pas dans master à partir du dernier commit de master. C'est ce que va faire (à partir de mybranch):

```
git rebase master
```

Pas besoin d'être en interactif ici, on veut les rejouer tels quels.

Si un conflit devait arriver, `git rebase` passera automatiquement en mode interactif.



# git en local - nettoyage

```
git checkout mybranch  
git rebase master
```

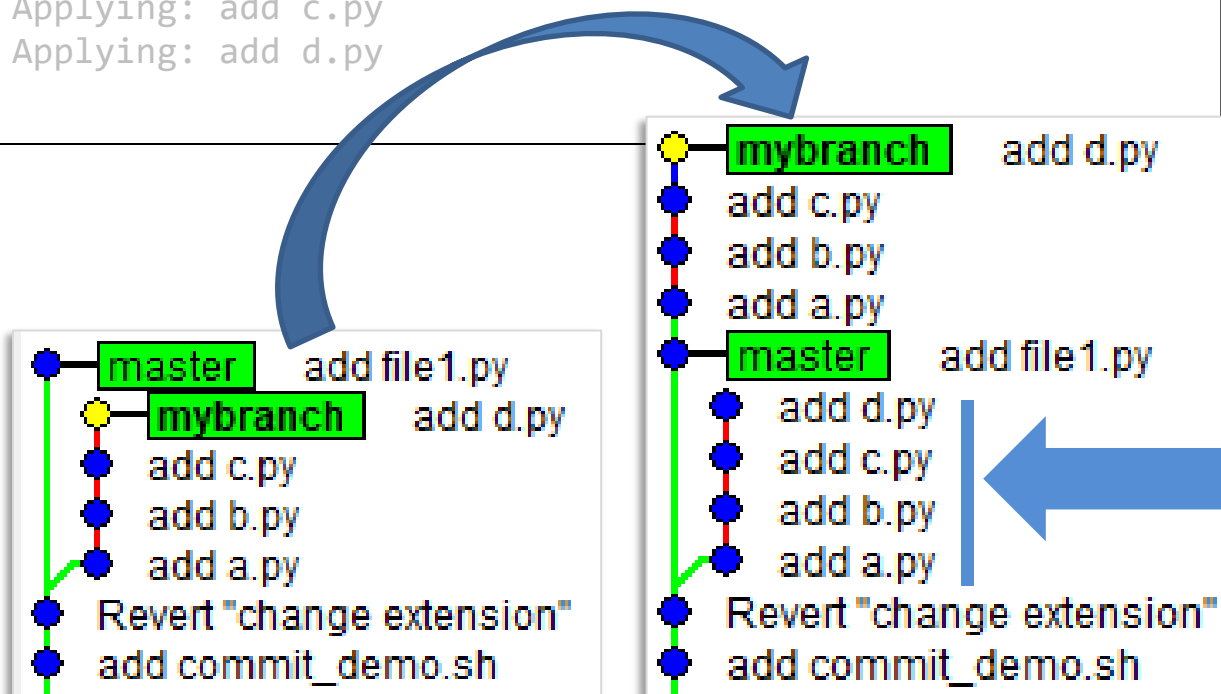
First, rewinding head to replay your work on top of it...

Applying: add a.py

Applying: add b.py

Applying: add c.py

Applying: add d.py



L'ancienne branche est toujours là mais elle n'est plus atteignable sans examiner le reflog.

Elle sera supprimée par le garbage collector quand le reflog sera périmé (90 jours par défaut).

Permet de rendre l'historique linéaire.

# Utiliser git

PARTIE 1  
Utiliser git en local

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int max)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        #pragma omp parallel for num_threads(nbt)
        for (int j=0; j<n; j++)
        {
            c[i*n+j] = a[i*n+j] + b[i*n+j];
        }
    }
}
```

## Derniers mots

```
double tstart = omp_get_wtime();
test.execute(nbt);
double tstop = omp_get_wtime();
double cpu = tstop-tstart;

OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

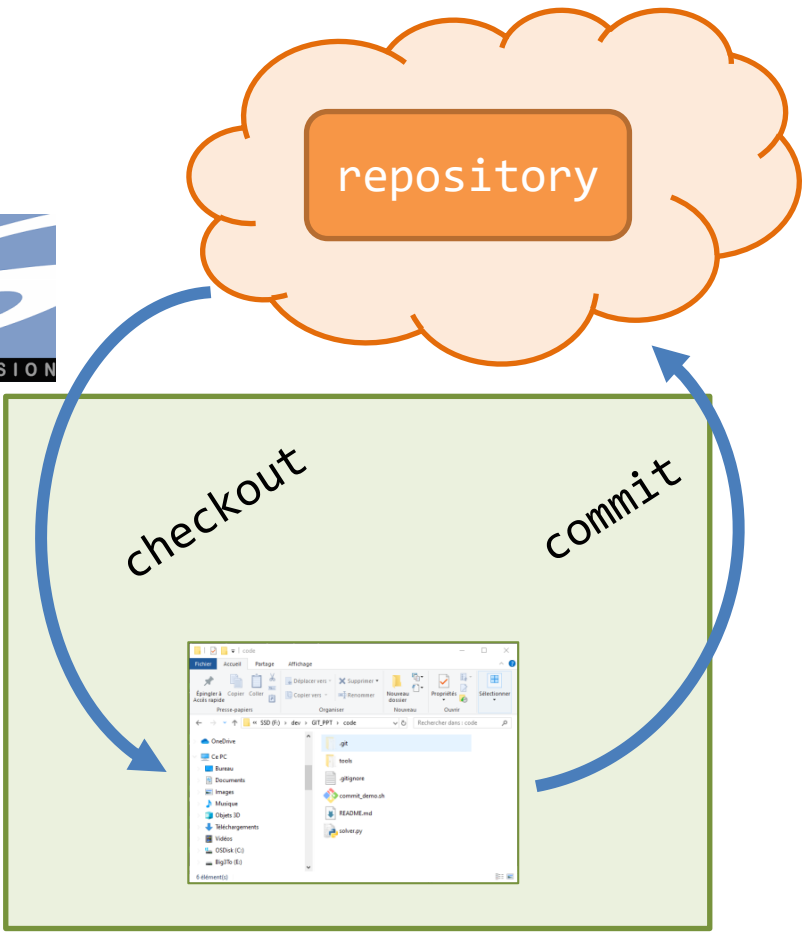
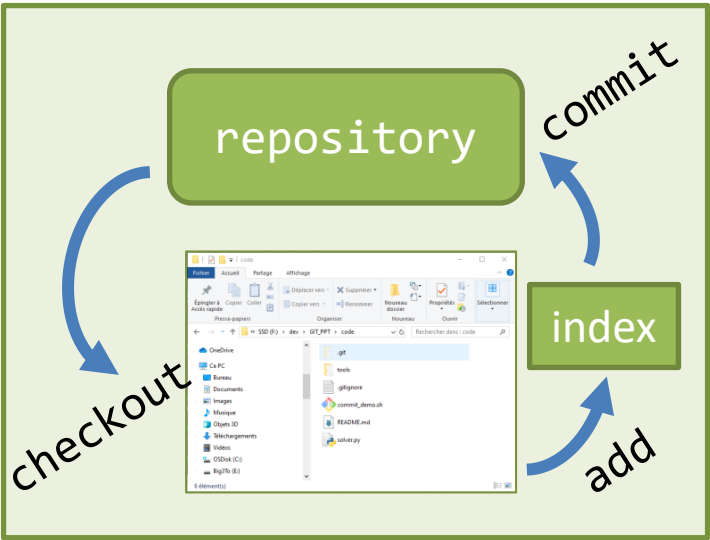
std::cout << res;
```



# git en local - derniers mots

## Comparaison de git et subversion (1/2)

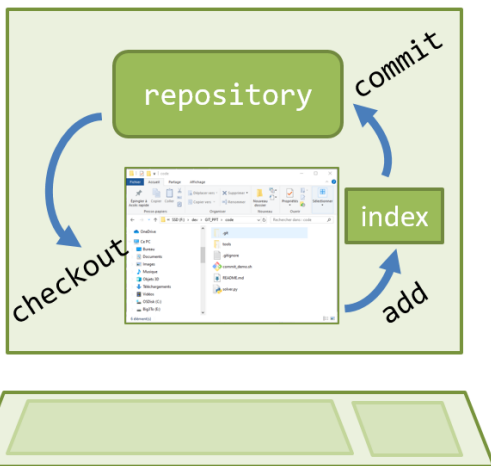
...pour les "anciens" développeurs





# git en local - derniers mots

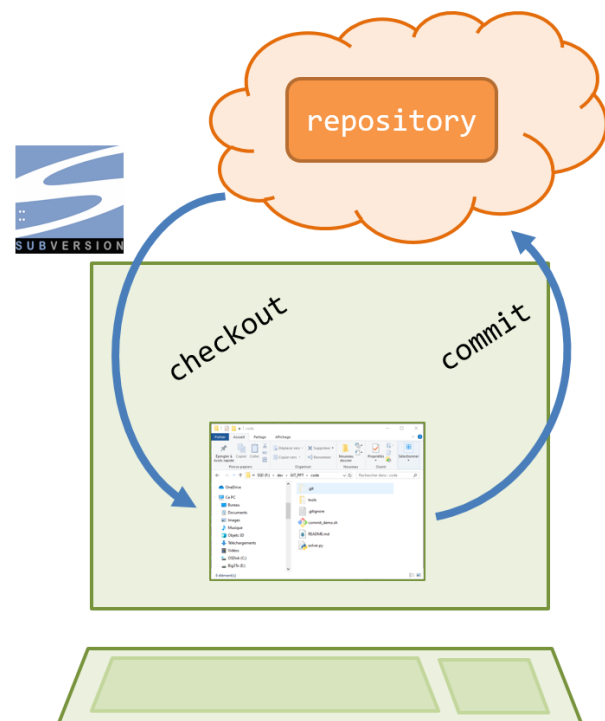
## Comparaison de git et subversion (2/2)



Par défaut,  
le repository SVN est sur le RESEAU  
(sur un serveur) et PARTAGE ;  
le repository git est LOCAL (sur  
votre PC) et PRIVE.

Les commandes checkout,  
commit ont le même nom et font  
la même chose !

git est infiniment plus riche que  
SVN pour gérer ses commits et  
merger des branches.  
Ceci en partie grâce à l'index qui  
permet de « préparer un commit »



Si votre repository git est LOCAL, comment collaborer? => voir PARTIE 2...





# git en local - derniers mots

## *Quelques commandes non présentées*

`git cherry-pick <commit ID>`

Applique un commit quelconque (c'est-à-dire les modifications) à la branche en cours.  
Opération inverse de revert

`git blame <file>`

Affiche le contenu d'un fichier où chaque ligne est préfixée par le dernier auteur de cette ligne.  
Utile pour savoir « qui a introduit un bug »

`git ls-files`

Liste les fichiers par catégorie (ceux gérés par git, ceux qui ne le sont pas, ceux qui ont été supprimés, etc.)

`git gc`

`git prune`

Nettoyage du repository et du reflog.  
Suppression des commits perdus et des branches mortes.

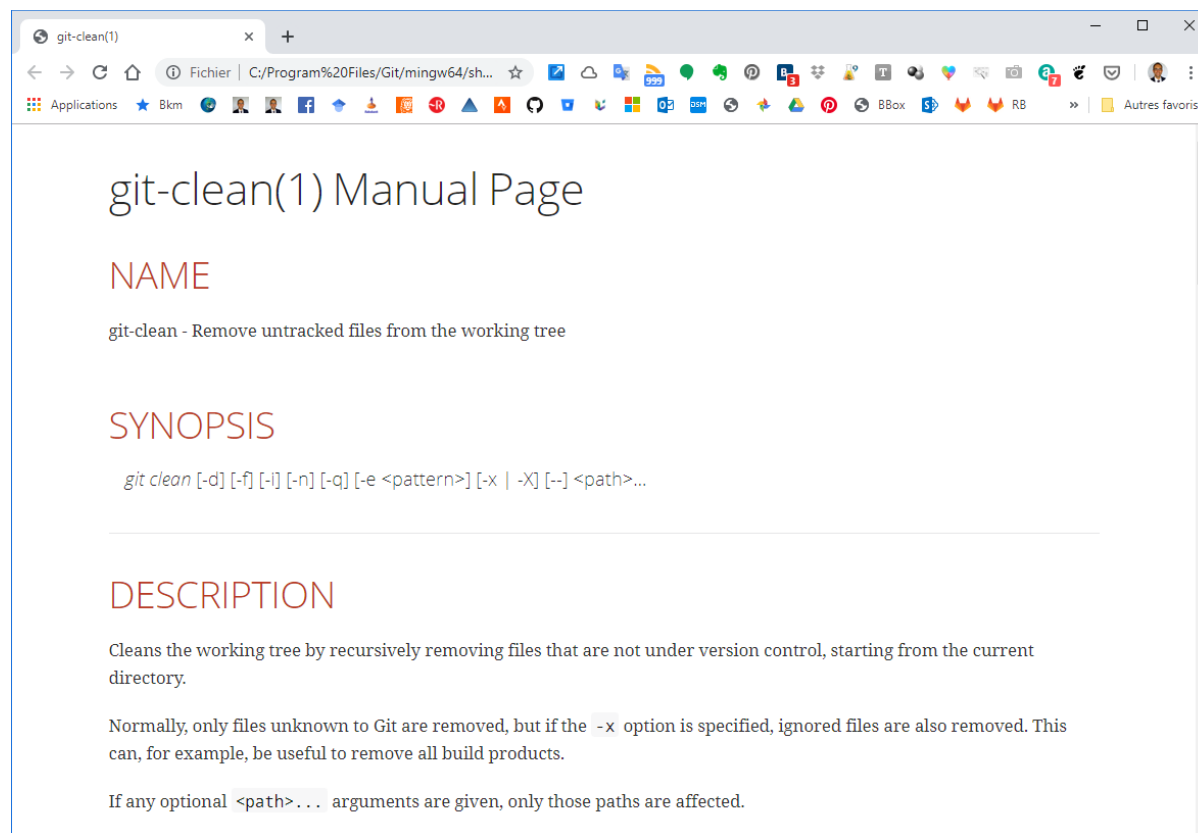


# git en local - derniers mots

## Obtenir de l'aide

Ajouter `--help` à la commande

`git clean --help`



**voir aussi:** <https://www.atlassian.com/git/tutorials>

# Utiliser git

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int tmax)
{
    #pragma omp parallel for num_threads(nbt)
```

```
(nbt)
```

## PARTIE 2

# Utiliser git avec un remote repo.

```
double tstop = omp_get_wtime();
double cpu = tstop-tstart;
```

```
OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));
```

```
std::cout << res;
```

# Utiliser git

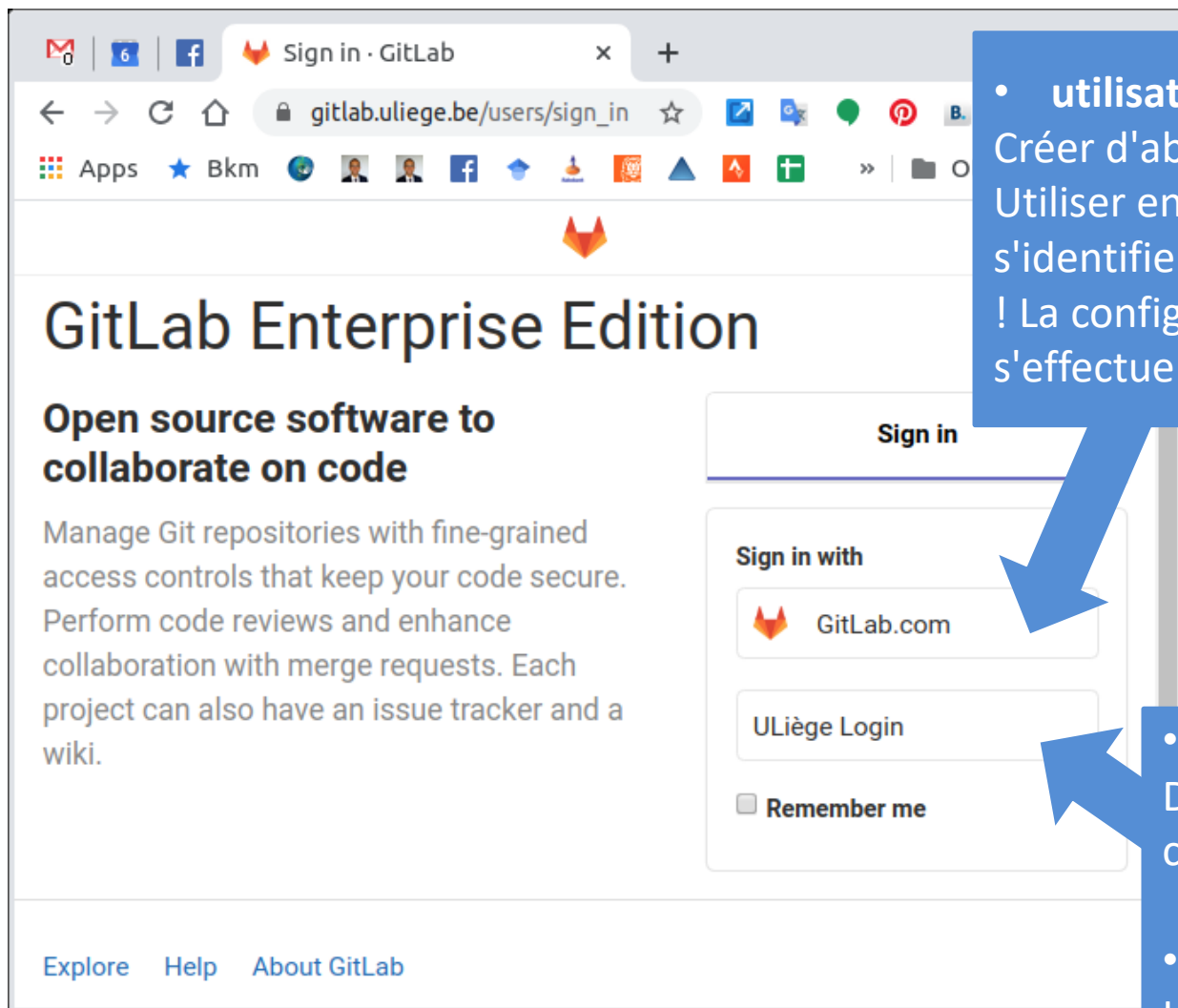
PARTIE 2  
Utiliser git avec un  
remote repo.

## Configurer GitLab de l'ULiège



# Configurer GitLab

**Première connexion à GitLab ULiège:** <https://gitlab.uliege.be/>



- **utilisateur externe:**

Créer d'abord un compte sur [gitlab.com](https://gitlab.com)  
Utiliser ensuite ce bouton pour s'identifier avec l'identifiant gitlab.com.  
! La configuration qui suit (clef ssh) s'effectue alors sur [gitlab.com](https://gitlab.com) !

- **personnel ULiège:**

Demander l'activation de votre compte uXXXXXX à F. Schoofs

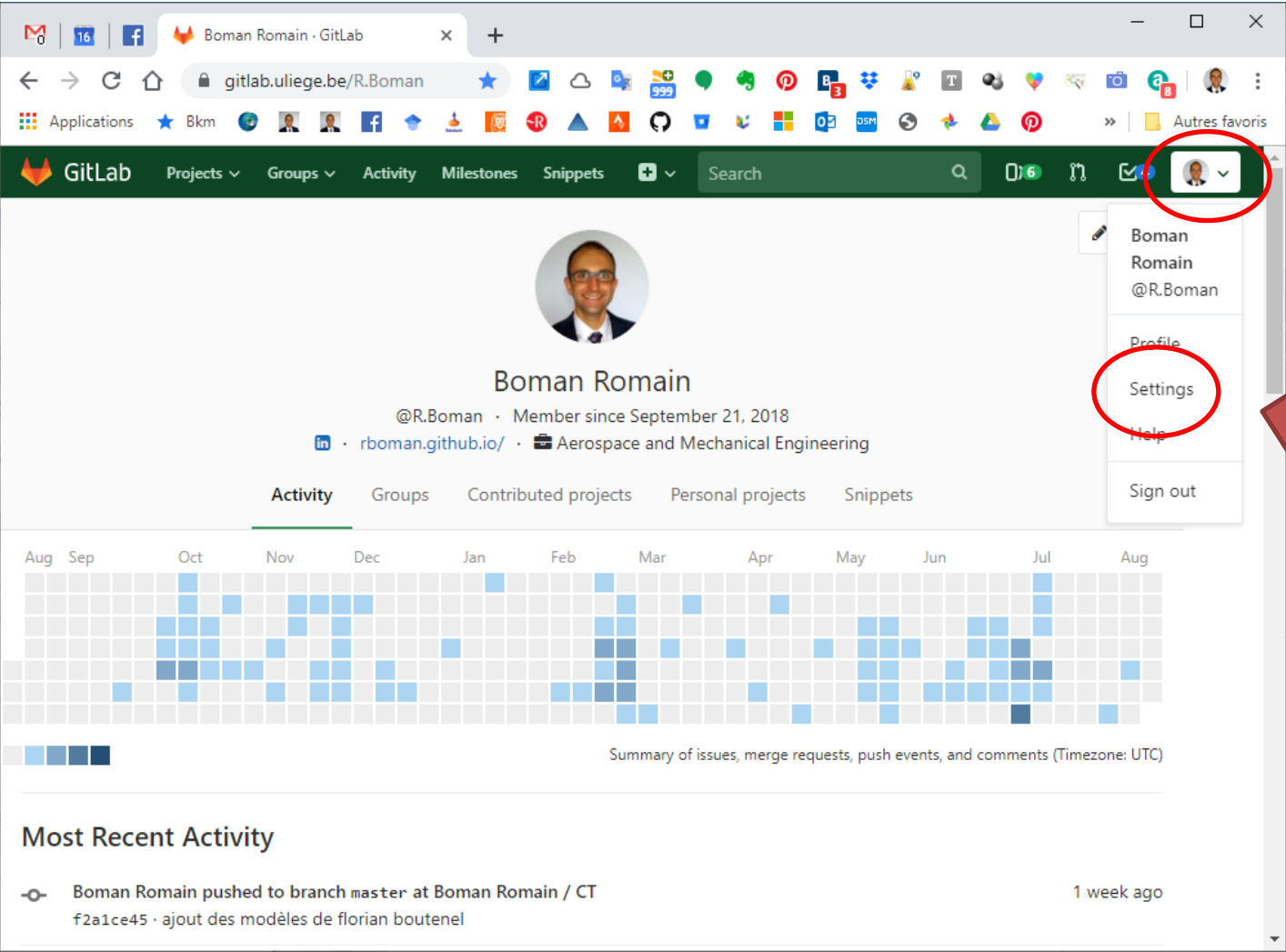
- **étudiants:**

Utilisez votre identifiant sXXXXX



# Configurerer GitLab

## Configuration du compte





# Configurer GitLab

## Configuration du compte – "clefs SSH"

The screenshot shows the GitLab web interface. The browser address bar displays 'gitlab.uliege.be/profile/keys'. The left sidebar contains a 'User Settings' menu with 'SSH Keys' highlighted by a red circle. The main content area is titled 'SSH Keys' and includes instructions on how to add a new SSH key. A blue box with an arrow points to the 'SSH Keys' menu item, and another blue box with an arrow points to the 'Add an SSH key' section.

Toute la procédure qui suit peut être réalisée à l'aide de "git bash", comme décrit dans les liens ci-dessous.

Contrairement à beaucoup d'autres serveurs git, la communication avec le GitLab de ULiège se fait **exclusivement par SSH**

Add an SSH key

To add an SSH key you need to [generate one](#) or use an [existing key](#).

Key

Paste your public SSH key, which is usually contained in the file '~/.ssh/ssh-rsa'. Don't use your private SSH key.

Typically starts with "ssh-rsa ..."

Si vous en avez déjà une, collez votre clef SSH publique ici, sinon lisez la suite...

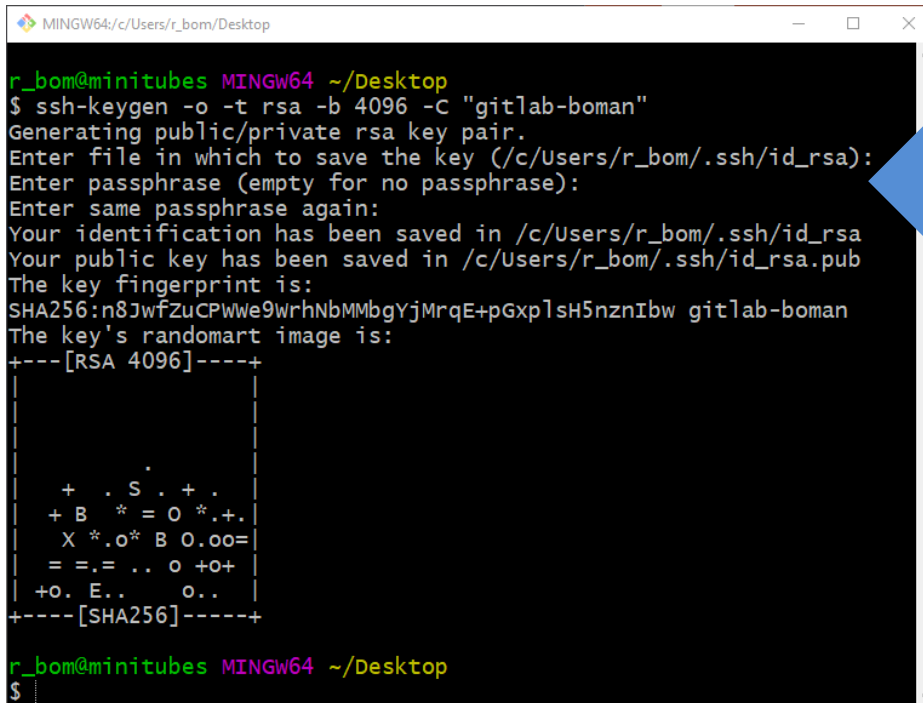


# Configurerer GitLab

## Configuration du compte – "clefs SSH"

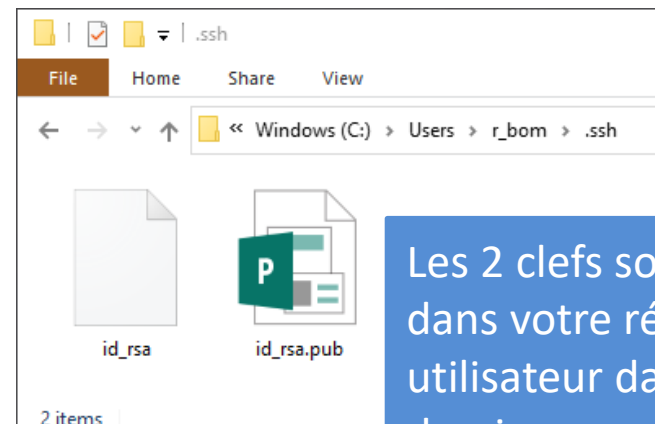
Ouvrez un git-bash et exécutez la commande:

```
ssh-keygen -o -t rsa -b 4096 -C "nom-de-votre-clef"
```



```
MINGW64/c/Users/r_bom/Desktop
r_bom@minitubes MINGW64 ~/Desktop
$ ssh-keygen -o -t rsa -b 4096 -C "gitlab-boman"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/r_bom/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/r_bom/.ssh/id_rsa
Your public key has been saved in /c/Users/r_bom/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:n8JwfZuCPWwe9WrhNbMMbgYjMrqE+pgXplSH5nznIbw gitlab-boman
The key's randomart image is:
+---[RSA 4096]-----+
|
|+ . S . + .
|+ B * = O *.+
|X *.o* B O.o=
|=.=. . O +O+
|+O. E.. O..
+---[SHA256]-----+
r_bom@minitubes MINGW64 ~/Desktop
$
```

Choisissez  
l'emplacement par  
défaut et ne mettez  
pas de mot de passe.



Les 2 clefs sont générées  
dans votre répertoire  
utilisateur dans un  
dossier nommé ".ssh"

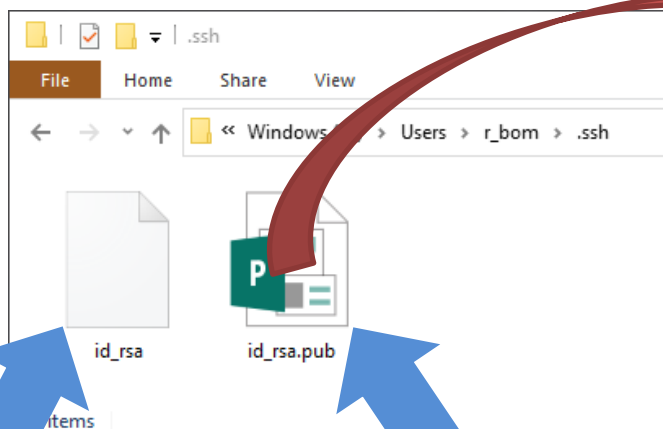




# Configurer GitLab

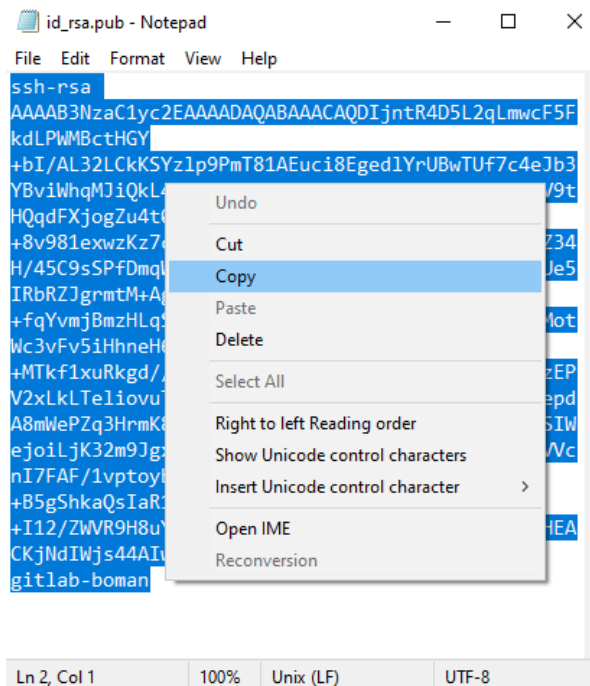
## Configuration du compte – "clefs SSH"

Ouvrez le fichier `id_rsa.pub` dans un éditeur de texte:



**Clef privée:** elle ne peut pas être partagée. Quiconque possède ce fichier peut se faire passer pour vous!

**Clef publique:** elle peut être donnée à n'importe qui voulant vérifier votre identité (GitLab)



si votre environnement n'est pas sécurisé, mettez un mot de passe à l'étape précédente.

Sélectionnez tout le texte (CTRL+A) et copiez-le dans le presse papier (CTRL-C)



# Configurerer GitLab

## Configuration du compte – "clefs SSH"

id\_rsa.pub - Notepad

File Edit Format View Help

ssh-rsa  
AAAAB3NzaC1yc2EAAAADAQABAAQADiJntR4D5L2qLmwcF5F  
kdLPWMBctHGY  
+bI/AL32LckSYzlp9PmT81AEuci8Eged1YrUBwTUf7c4eJb3  
YBviWhqMJiQkL  
HQqdFXjogZu4t  
+8v981exwzKz7  
H/45C9sSPfDmq  
IRbRZJgrmtM+A  
+fqYvmjBmzHLq  
Wc3vFv5iHhneH  
+MTkf1xuRkgd/  
V2xLkLTeliovu  
A8mWePZq3HrmK  
ejoilJk32m9Jg  
nI7FAF/1vptoy  
+B5gShkaQsIaR  
+I12/ZWVR9H8u  
CKjNdIWjs44AI  
gitlab-boman

Undo  
Cut  
Copy  
Paste  
Delete  
Select All  
Right to left Reading order  
Show Unicode control characters  
Insert Unicode control character  
Open IME  
Reconversion

SSH Keys

SSH keys allow you to establish a secure connection between your computer and GitLab.

Add an SSH key

To add an SSH key you need to generate one or use an existing key.

Key

Paste your public SSH key, which is usually contained in the file '~/.ssh/id\_ed25519.pub' or '~/.ssh/id\_rsa.pub' and begins with 'ssh-ed25519' or 'ssh-rsa'. Don't use your private SSH key.

4H/45C9sSPfDmqWoMAyuSw3VuCPVzT3IY3IC/NBaYz1VclAUe5IRbRZJgrmtM+AgoMrWWa  
ags+fqYvmjBmzHLqS1px4aaUk0VpuPCSk5Tiz7WJLqnn4VJuMotWc3vFv5iHhneH6OwxWJf  
1mhPLirT35ySmO+MTkf1xuRkgd//FhrKkPlrznJRSE7GJQtv1RAfMn7XB5IAzEPV2xLkLTeliovu  
TkyWwOqyeBFkR7s1MyjhOBK98oRoHuMSPePdA8mWePZq3HrmK88IfDQ9/19ARDZ/daUy  
61YAsI7Musc8oRSIWejoilJk32m9Jgx0prHFZ59laTK37qTokB70xPUUaesUKQhVVcnI7FAF/1vp  
toybDm3licGr+pt6LpD+B5gShkaQslaR13roKWU+I12/ZWVR9H8uY17ry1/1hsnwZWvkF676  
4RRxICkAsOKziHEACKjNdIWjs44AlwJm+rLg6T5GtZk1y3zfuvt0MYCD9w== gitlab-boman

Title

gitlab-boman

Name your individual key via a title

Add key

Collez la clef (CTRL-V) dans GitLab et cliquez sur "add key".



# Configurerer GitLab

## Configuration du compte – "clefs SSH"

Dans une fenêtre git-bash, vous pouvez tester si GitLab accepte votre clef:

```
ssh -T git@gitlab.uliege.be
```

répondez « yes »

```
MINGW64:/c/Users/r_bom/Desktop  
r_bom@minitubes MINGW64 ~/Desktop  
$ ssh -T git@gitlab.uliege.be  
The authenticity of host 'gitlab.uliege.be (139.165.47.38)' can't be established  
.  
ECDSA key fingerprint is SHA256:4DyJeec4s+THW2EOZw5xjGgk6ZUcgmWjCcX33nVep7g.  
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes  
Warning: Permanently added 'gitlab.uliege.be,139.165.47.38' (ECDSA) to the list  
of known hosts.  
Welcome to GitLab, @R.Boman!
```

Si vous recevez ce type de message de bienvenue, tout est configuré correctement!



# Configurer GitLab



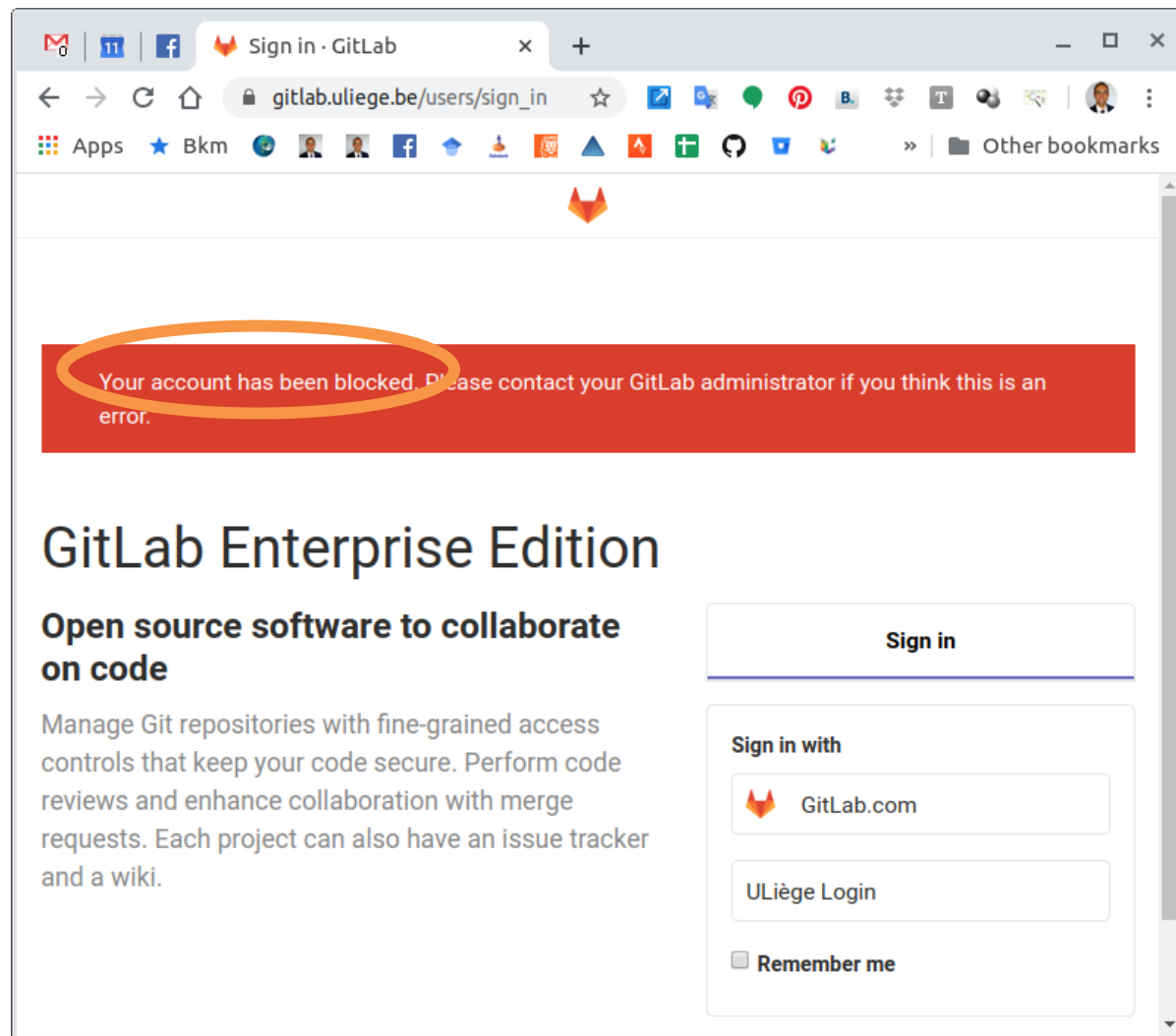
## Attention!

Le SeGI a configuré GitLab pour qu'il bloque très vite un utilisateur qui se trompe de mot de passe plusieurs fois d'affilée.

Si ça ne marche pas, vérifiez toute la procédure avant de réessayer plusieurs fois la même chose!

Bloqué?

<https://sam.segi.uliege.be/>



# Utiliser git

PARTIE 2  
Utiliser git avec un  
remote repo.

## Utiliser GitLab

pour collaborer avec d'autres et sauvegarder son travail



# Utiliser GitLab

## Introduction

git permet de "partager" son repository avec d'autres développeurs et collaborer effacement avec eux.

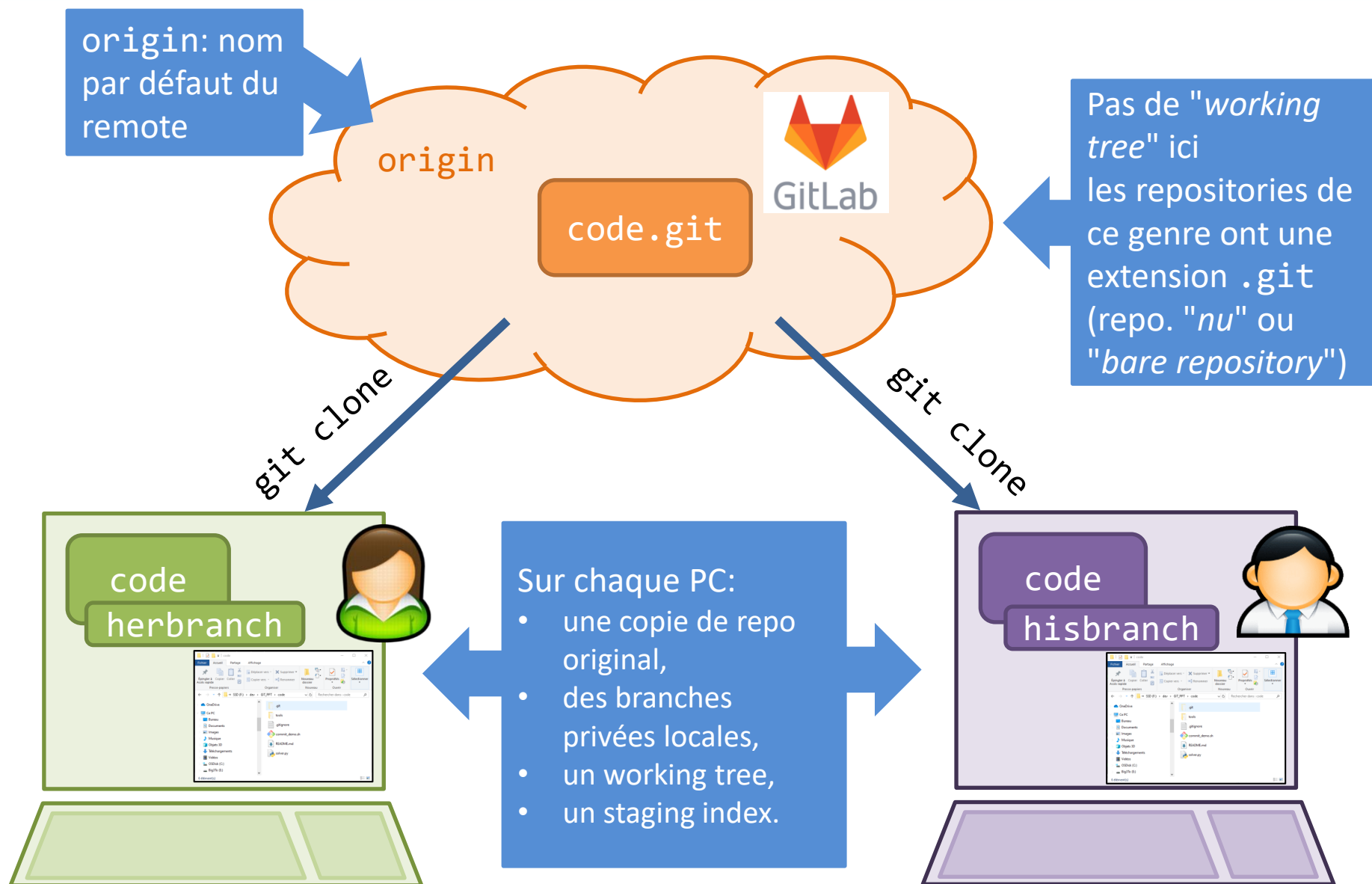
## Comment?

- Simplement en le "copiant" sur un serveur sur le réseau tel GitLab ("*remote*") .
- Aucun utilisateur n'est associé à cette copie ("*bare repository*") qui devient une référence pour plusieurs développeurs.
- Chaque nouveau développeur copie (ou "clone") ce repository (`git clone`).
- Le repository cloné va garder en mémoire d'où il provient (pointeur `origin`).
- Les clones des branches du repository (`origin`) pourront être enrichies localement par de nouveaux commits (voir PARTIE 1).
- A ces opérations locales s'ajoutent les 2 nouvelles commandes `fetch` et `push` qui, respectivement, met à jour la copie locale du repository source et qui envoie de nouveaux commits vers le repository source.

De manière plus générale, **n'importe quel repository peut jouer le rôle de remote**. On parle de système DISTRIBUE (et non CENTRALISE comme SVN )



# Utiliser GitLab



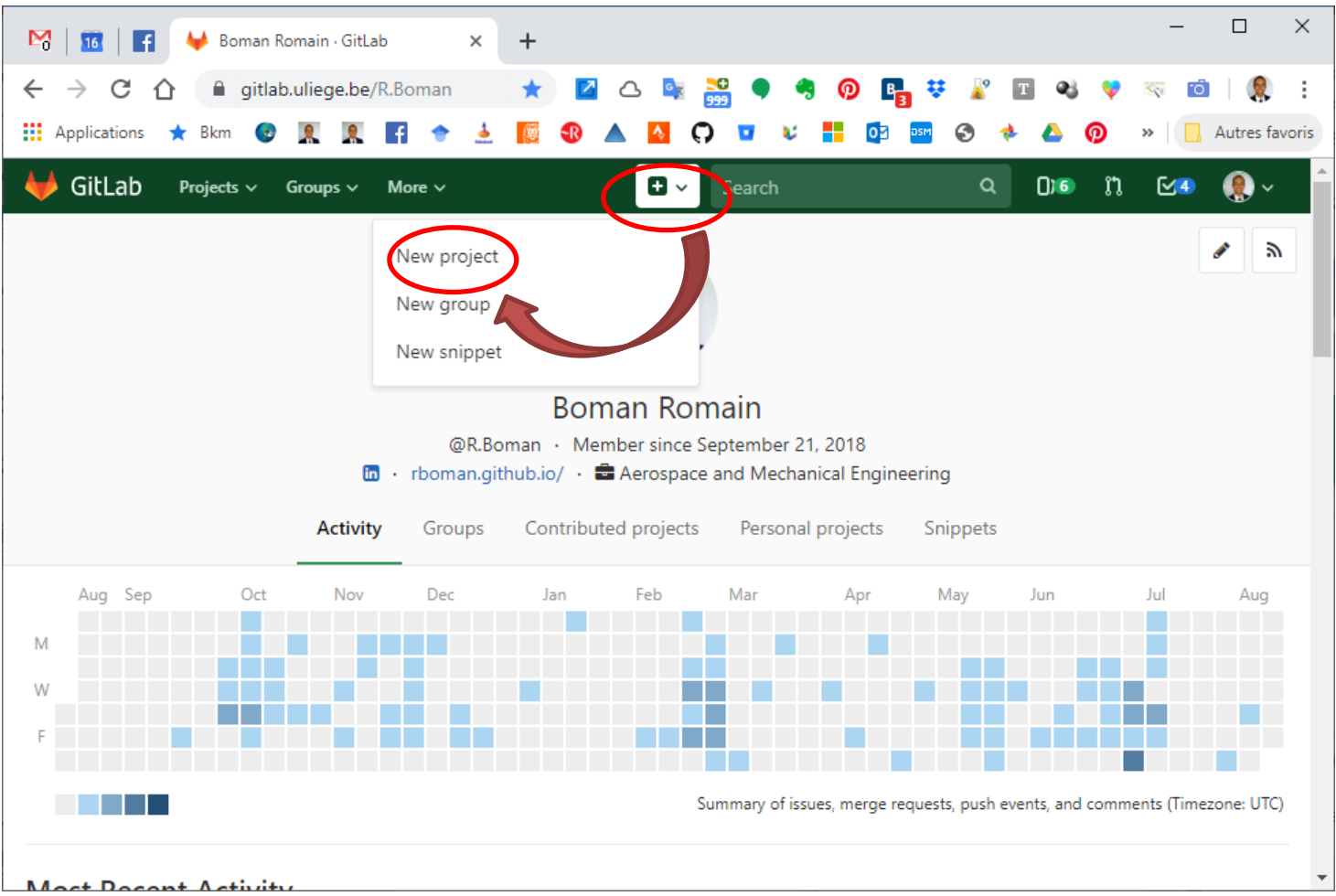




# Utiliser GitLab

## Envoyer un repository local vers GitLab (1/6)

On crée un nouveau "projet" GitLab







# Utiliser GitLab

## Envoyer un repository local vers GitLab (2/6)

Blank project

Create from template

Import project

CI/CD for external repo

Project path

Project name

https://gitlab.uliege.be/

R.Boman

code

Want to house several dependent projects under the same namespace? Create a group

Project description (optional)

Description format

Visibility Level ?

☒ Private

Project access must be granted explicitly to each user.

☐ Internal

The project can be accessed by any logged in user.

☐ Public

The project can be accessed without any authentication.

☐ Initialize repository with a README

Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

Create project

Cancel



# Utiliser GitLab

## Envoyer un repository local vers GitLab (3/6)

GitLab

Projects ▾

Groups ▾

More ▾

▾

This project

Search

6

4

C

Boman Romain > code > Details

Project 'code' was successfully created.

code

Star

0

SSH

git@gitlab.uliege.be:R.Boman/code.g

Global ▾

The repository for this project is empty

If you already have files you can push them using the [command line instructions](#) below.

adresse internet  
du repository

instructions!



# Utiliser GitLab

## ***Envoyer un repository local vers GitLab (4/6)***

(on suit les instructions de GitLab aveuglément - on comprendra plus tard)

```
cd code
```

```
git remote add origin git@gitlab.uliege.be:R.Boman/code.git
```

```
git push -u origin --all
```

```
Enumerating objects: 38, done.
```

```
Counting objects: 100% (38/38), done.
```

```
Delta compression using up to 12 threads
```

```
Compressing objects: 100% (24/24), done.
```

```
Writing objects: 100% (38/38), 3.65 KiB | 467.00 KiB/s, done.
```

```
Total 38 (delta 6), reused 32 (delta 4)
```

```
To gitlab.uliege.be:R.Boman/code.git
```

```
* [new branch]      master -> master
```

```
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

```
git push -u origin --tags
```

```
Total 0 (delta 0), reused 0 (delta 0)
```

```
To gitlab.uliege.be:R.Boman/code.git
```

```
* [new tag]         v1.0 -> v1.0
```



# Utiliser GitLab

## Envoyer un repository local vers GitLab (5/6)

Retour sur GitLab et actualiser la page (F5):

master

code / +

History

Find file

Web IDE

Revert "change extension"  
Romain Boman authored 1 day ago

df7cdf31

Name	Last commit	Last update
tools	new code structure	3 days ago
.gitignore	Merge branch 'mybranch'	1 day ago
README.md	Revert "change extension"	22 hours ago
commit_demo.sh	add commit_demo.sh	22 hours ago
solver.py	better input	2 days ago

Les fichiers sont  
visibles en ligne!



# Utiliser GitLab

## Envoyer un repository local vers GitLab (6/6)

GitLab propose toutes sortes d'outils graphiques pour analyser les fichiers, les commits, visualiser des différences, etc.

Par exemple, "Graph" est similaire à gitk

The screenshot shows the GitLab web interface for a project named 'code' by user 'Boman Romain'. The left sidebar contains a navigation menu with options: 'code', 'Project', 'Repository', 'Files', 'Commits', 'Branches', 'Tags', 'Contributors', 'Graph', 'Compare', 'Charts', 'Locked Files', 'Issues', and 'Merge Requests'. The 'Files' and 'Graph' items are circled in red. A blue arrow points from the text 'Par exemple, "Graph" est similaire à gitk' to the 'Graph' item in the sidebar. The main content area displays the commit graph for the 'master' branch. It includes a search bar for 'Git revision' and a checkbox for 'Begin with the selected commit'. The graph shows a vertical timeline of commits with dates (Aug 15, 14, 13) and commit messages such as 'Revert "change extension"', 'add commit\_demo.sh', 'Merge branch "mybranch"', 'add .tmp to .gitignore', 'add .bak to gitignore', 'Merge branch "mybranch"', 'better input', 'change extension', 'new code structure', 'better output', 'add python header', and 'add file solver.py'. The 'master' branch is indicated at the top of the graph, and a 'v1.0' tag is visible at the bottom.



# Utiliser git

PARTIE 2  
Utiliser git avec un  
remote repo.

## Cloner un projet git

```
void mxv(int m, int n, double *a, double *b, double *c, int max)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        #pragma omp parallel for num_threads(nbt)
        for (int j=0; j<n; j++)
        {
            c[i*n+j] = a[i*n+j] + b[i*n+j];
        }
    }

    double tstart = omp_get_wtime();
    test.execute(nbt);
    double tstop = omp_get_wtime();
    double cpu = tstop-tstart;

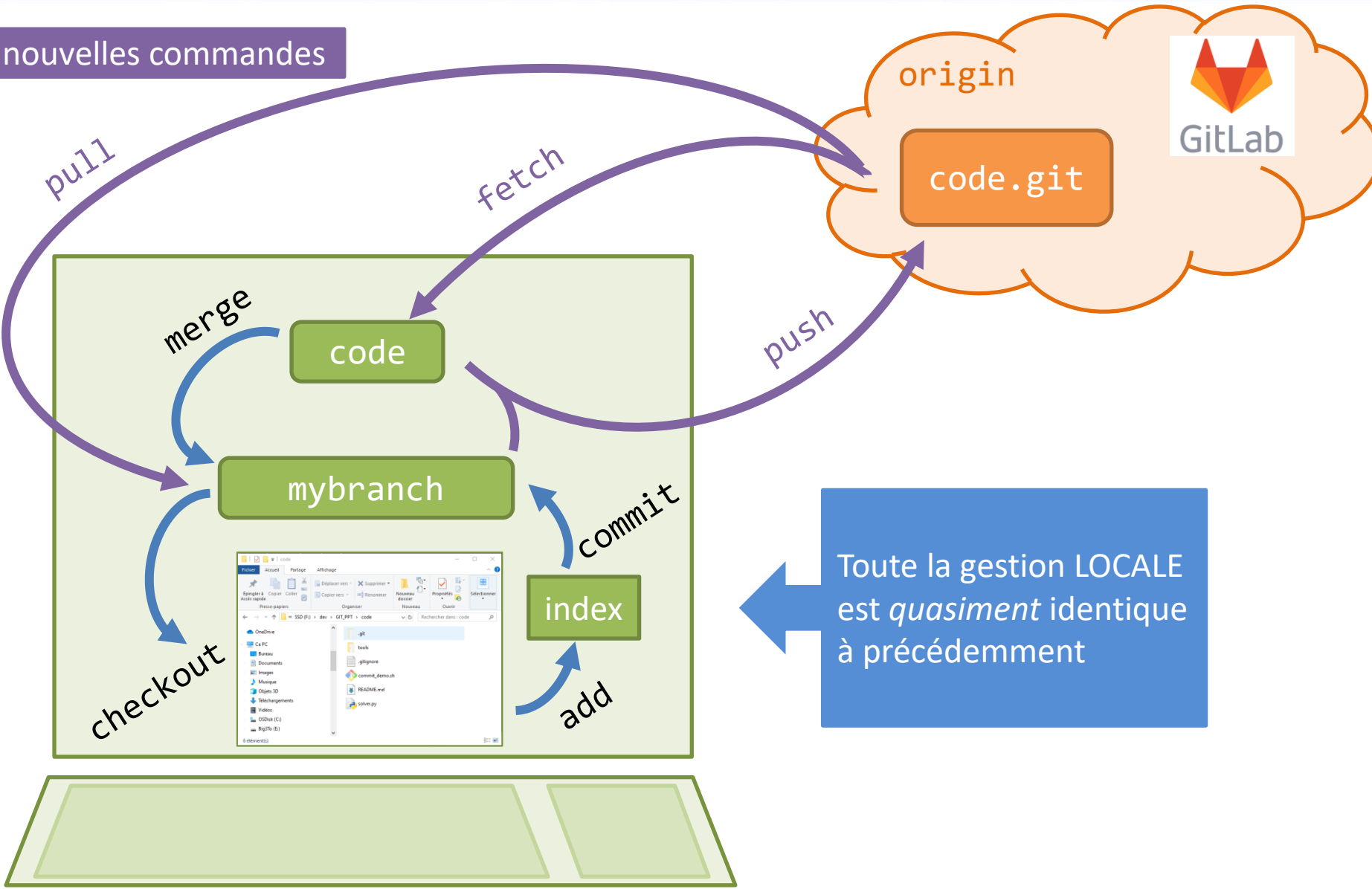
    OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

    std::cout << res;
}
```



# Cloner un projet git

nouvelles commandes






# Cloner un projet git

## ***Récupération d'un clone (1/2)***

[supprimez d'abord le dossier qui vient d'être envoyé sur GitLab]

```
git clone git@gitlab.uliege.be:R.Boman/code.git
Cloning into 'code'...
remote: Counting objects: 38, done.
remote: Compressing objects: 100% (28/28), done.
remote: Total 38 (delta 7), reused 0 (delta 0)
Receiving objects: 100% (38/38), done.
Resolving deltas: 100% (7/7), done.
```

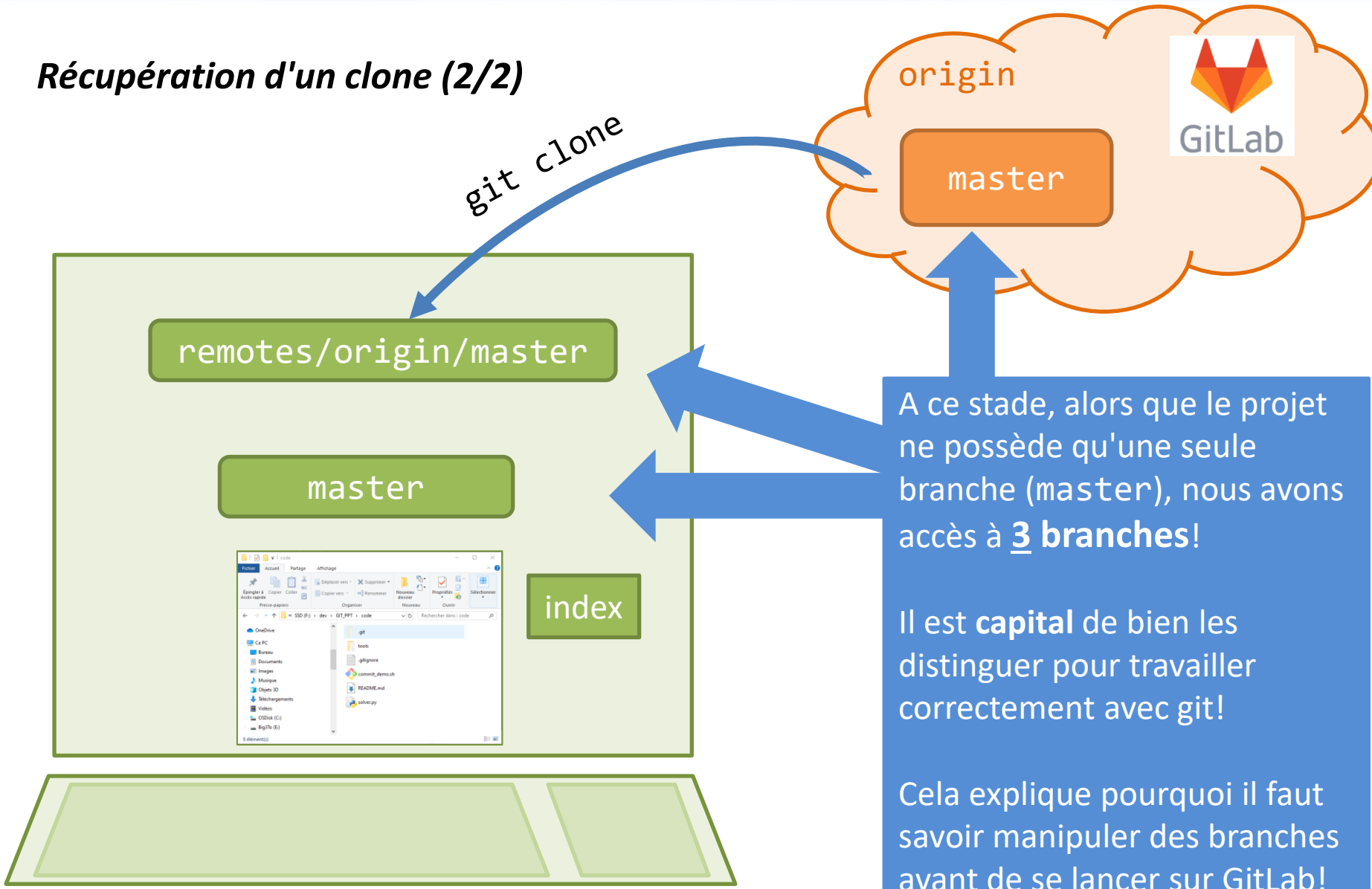
Remarque: si vous venez d'envoyer votre repository local sur GitLab en suivant cette présentation, vous pouvez toujours continuer à utiliser votre repository et vous n'avez pas besoin de faire un clone (contrairement à SVN où "svn checkout" est obligatoire après `svn import` ).





# Cloner un projet git

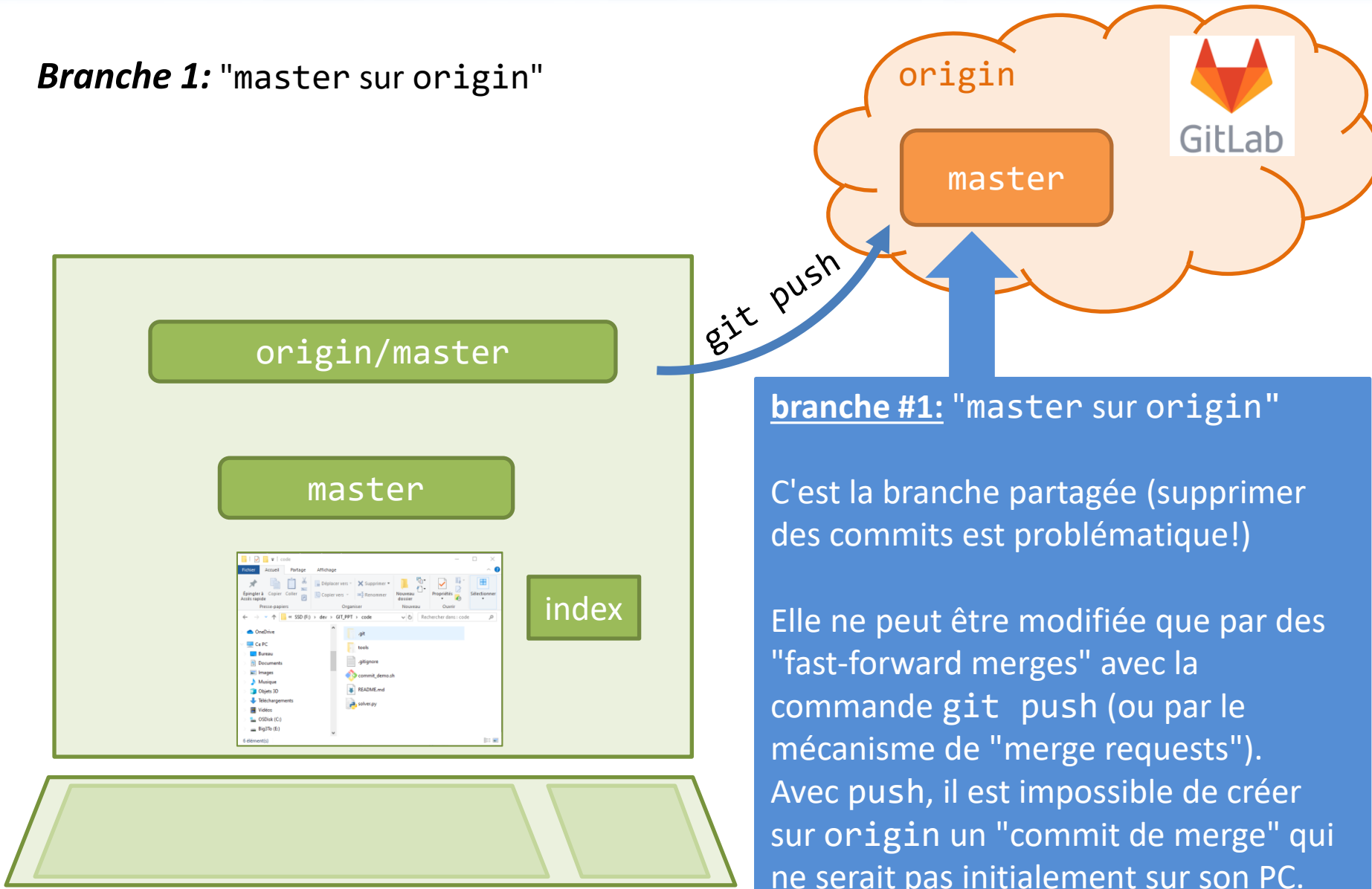
## Récupération d'un clone (2/2)





# Cloner un projet git

**Branche 1:** "master sur origin"



branche #1: "master sur origin"

C'est la branche partagée (supprimer des commits est problématique!)

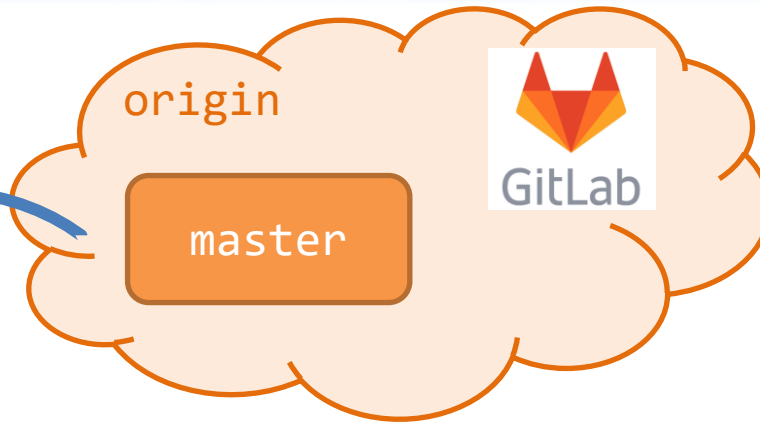
Elle ne peut être modifiée que par des "fast-forward merges" avec la commande `git push` (ou par le mécanisme de "merge requests"). Avec push, il est impossible de créer sur origin un "commit de merge" qui ne serait pas initialement sur son PC.



# Cloner un projet git

**Branche 2:** "origin/master"  
(ou "remotes/origin/master")

git fetch



branche #2: "origin/master"

C'est la copie locale de "master sur origin" (à un moment donné)

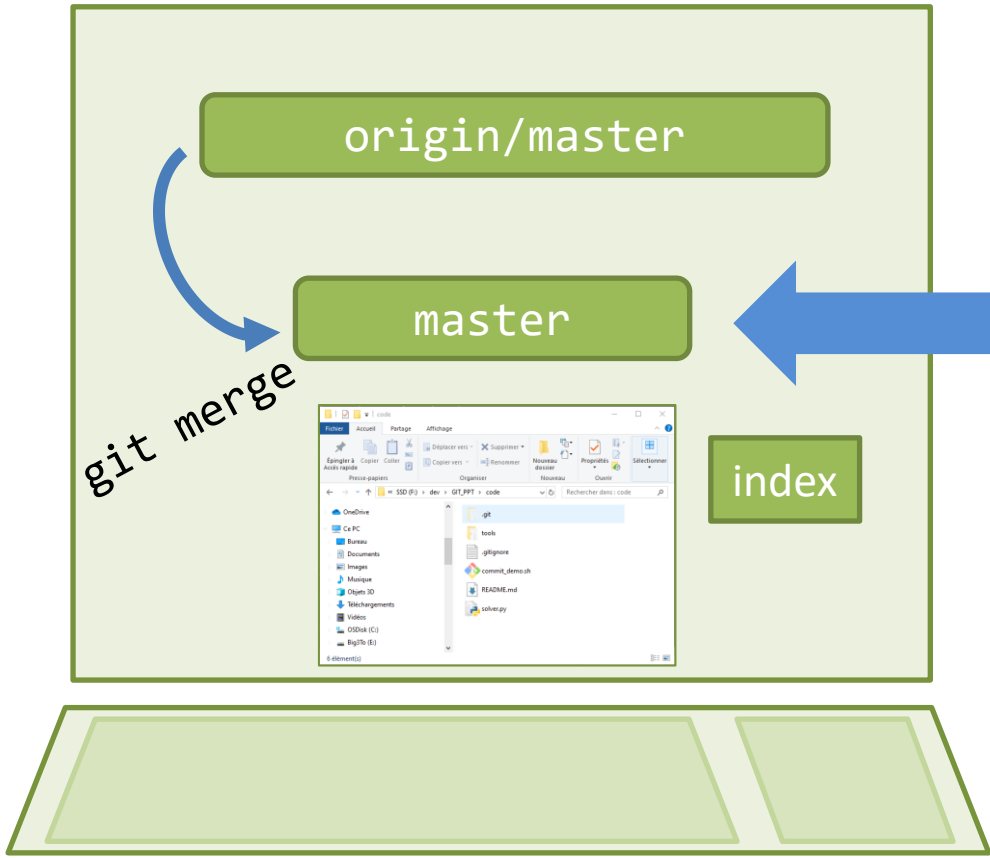
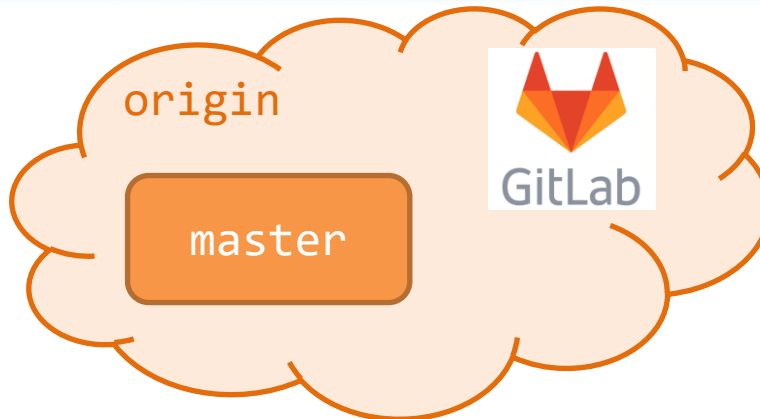
Elle permet de travailler hors-ligne.

Elle peut être "en retard" par rapport à "master sur origin". Dans ce cas, elle est synchronisée avec la commande fetch.



# Cloner un projet git

**Branche 3:** branche locale "master"



branche #3: master

C'est la branche locale **privée** de travail. On y travaille comme vu précédemment. En particulier on y merge les nouveautés reçues de origin/master.

Elle est initialisée à origin/master  
C'est une "*tracking branch*": elle sait qu'elle est liée à master sur origin (facilite certaines commandes).

# Utiliser git

PARTIE 2  
Utiliser git avec un  
remote repo.

Une seule branche sur origin

```
void mxv(int m, int n, double *a, double *b, double *c, int max)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        #pragma omp parallel for num_threads(nbt)
        for (int j=0; j<n; j++)
        {
            c[i*n+j] = a[i*n+j] + b[i*n+j];
        }
    }

    double tstart = omp_get_wtime();
    test.execute(nbt);
    double tstop = omp_get_wtime();
    double cpu = tstop-tstart;

    OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

    std::cout << res;
}
```



# Une seule branche sur origin

**git ne propose pas une manière unique de travailler avec les branches.** Cette flexibilité est une barrière à l'apprentissage de git.

Il est nécessaire de définir préalablement une façon de travailler (workflow) qui va dépendre:

- du nombre de développeurs,
- de la taille du projet.

Dans cette section, on propose le **workflow le plus simple**:  
origin possède 1 seule branche: master

Ce workflow peut être utile pour des petits projets où le nombre de développeurs est faible.

Par exemple: pour **un seul développeur qui code à plusieurs endroits** (son desktop, son laptop, une station de calcul).



Ce workflow correspond **exactement** à ce qui est généralement fait avec SVN (pour lequel gérer des branches est un cauchemar)



# Une seule branche sur origin

## Exemple pratique

```
git clone git@gitlab.uliege.be:R.Boman/code.git
cd code/
```

```
git status
```

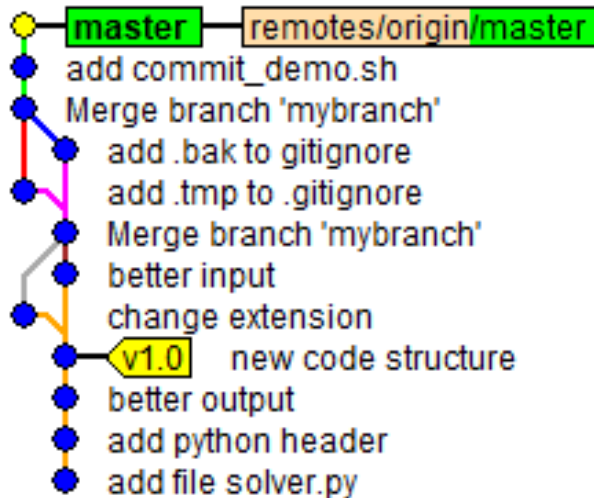
```
On branch master
```

```
Your branch is up to date with 'origin/master'.
```

```
nothing to commit, working tree clean
```

```
gitk --all
```

nouveau message utile (dû au fait que master est une tracking-branch): master et origin/master pointe vers le même commit!



gitk indique, en effet, que les 2 branches sont synchronisées.

Remarque: la branche "master sur origin" n'est pas affichée.

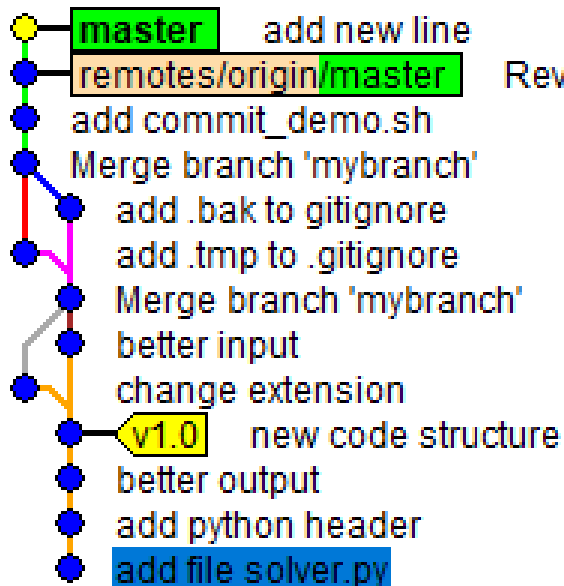


# Une seule branche sur origin

```
vi README.md
git add README.md
git commit -m "add new line"
[master b129ef5] add new line
1 file changed, 1 insertion(+)
```

```
gitk --all
```

on modifie le code et  
on crée un nouveau  
commit



Revert "change extension"

les 2 branches ne sont  
plus synchronisées





# Une seule branche sur origin

## ***On veut envoyer notre nouveau commit sur origin (1/2)***

Cela se fait par la commande push:

(remarque: La commande push vers master nécessite d'être "Maintainer" du repository sur GitLab).

```
git push
```

```
To gitlab.uliege.be:R.Boman/code.git
```

```
! [rejected]          master -> master (non-fast-forward)
```

```
error: failed to push some refs to 'git@gitlab.uliege.be:R.Boman/code.git'
```

```
hint: Updates were rejected because the tip of your current branch is behind
```

```
hint: its remote counterpart. Integrate the remote changes (e.g.
```

```
hint: 'git pull ...') before pushing again.
```

```
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```



git nous signale que l'opération n'est pas possible: master a évolué sur origin

Autrement dit, quelqu'un a été plus rapide que nous et a déjà envoyé un/des commit(s) sur GitLab. Autre cas de figure: on a codé le w-e sur un autre PC et on n'a pas récupéré les commits du w-e avant de coder sur notre PC au bureau.



# Une seule branche sur origin

***On veut envoyer notre nouveau commit sur origin (2/2)***

On aimerait donc synchroniser `origin/master` sur `master`

Une suite de commandes du type:

```
git checkout origin/master  
git merge master
```

ne fonctionne pas! `origin/master` ne peut pas être modifié sans accéder à `origin`.



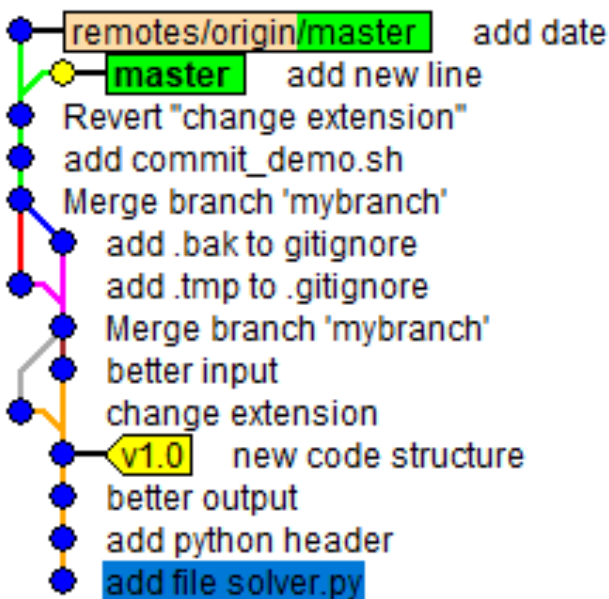
# Une seule branche sur origin

## Mise à jour de origin/master:

### git fetch

```
remote: Counting objects: 3, done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 3 (delta 1), reused 0 (delta 0)  
Unpacking objects: 100% (3/3), done.  
From gitlab.uliege.be:R.Boman/code  
    df7cdf3..41250f8  master    -> origin/master
```

- téléchargement des nouveaux commits
- origin/master se synchronise sur "master sur origin" qui a changé



Les 2 branches ont divergé

Il faut donc fusionner les branches master et origin/master avec les techniques vues précédemment (merge ou rebase).

C'est du travail local.



# Une seule branche sur origin

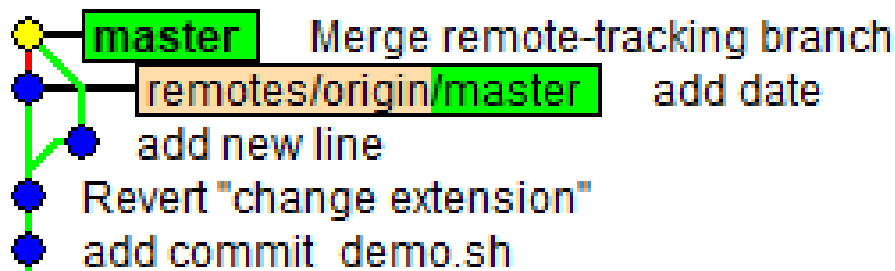
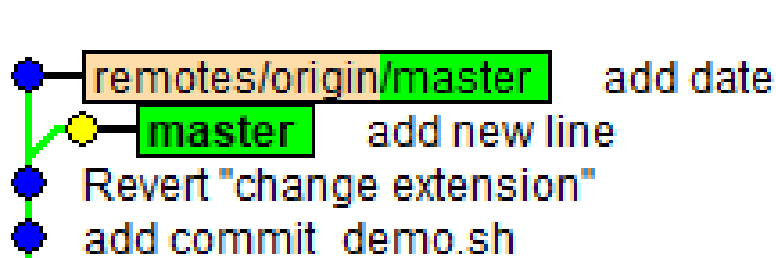
## Méthode #1: Fusion via "git merge" (méthode standard)

```
git merge origin/master
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.

vi README.md
git add README.md
git commit
[master 18dff11] Merge remote-tracking branch 'origin/master'
```

conflit...

et résolution  
comme  
précédemment



```
git push
```

```
git fetch  
git merge origin/master } git pull
```



# Une seule branche sur origin

## Méthode #2: Fusion via "git rebase" (pas conseillé aux débutants)

```
git rebase origin/master
```

First, rewinding head to replay your work on top of it...

[...]

CONFLICT (content): Merge conflict in README.md

error: Failed to merge in the changes.

[...]

```
vi README.md
```

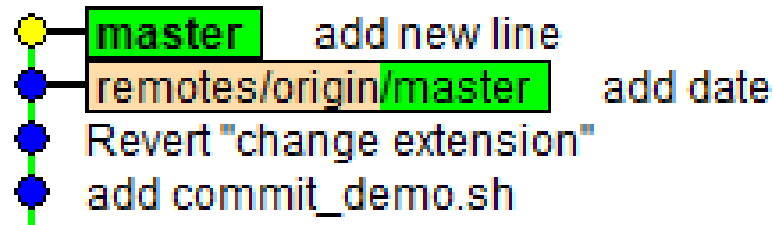
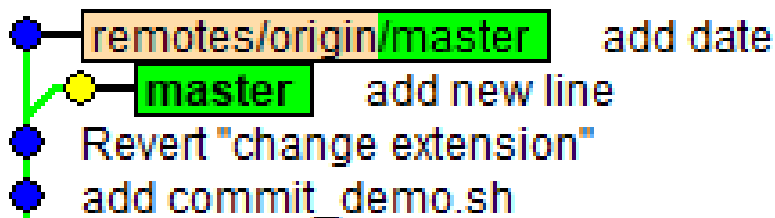
```
git add README.md
```

```
git rebase --continue
```

Applying: add new line

même conflit que  
dans le cas  
précédent...

historique "plus propre"  
(linéaire),  
mais on a perdu la  
chronologie du travail!

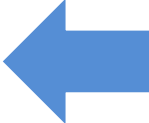




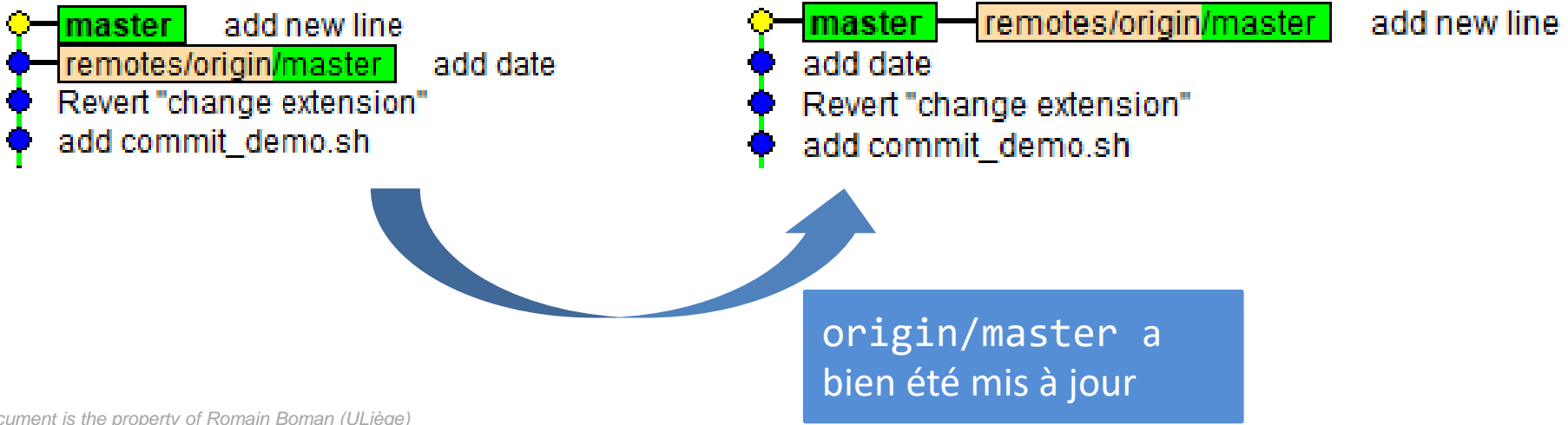
# Une seule branche sur origin

## Nouvel essai de "git push"

```
git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 333 bytes | 83.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To gitlab.uliege.be:R.Boman/code.git
 41250f8..8cc4b86  master -> master
```



tout se passe bien  
cette fois





# Une seule branche sur origin

## *Note pour les utilisateurs de SVN*

Equivalence pour "workflow 1 branche"

svn checkout	→	git clone
svn update	→	git pull (git fetch + git merge)
svn commit	→	git add git commit git push

Les commandes git manipulent 2 repositories!  
Elles font donc plus que leur "équivalent SVN"

Même pour ce workflow simple, git est supérieur à SVN pour les raisons suivantes:

- **convivialité**: interface web GitLab,
- **flexibilité**: possibilité d'utiliser des branches locales même s'il n'y en a pas sur origin,
- **backup**: le repository local est une copie intégrale de origin.

# Utiliser git

PARTIE 2  
Utiliser git avec un  
remote repo.

Plusieurs branches sur origin

```
void mxv(int m, int n, double *a, double *b, double *c, int max)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        #pragma omp parallel for num_threads(nbt)
        for (int j=0; j<n; j++)
        {
            c[i*n+j] = a[i*n+j] + b[i*n+j];
        }
    }

    double tstart = omp_get_wtime();
    test.execute(nbt);
    double tstop = omp_get_wtime();
    double cpu = tstop-tstart;

    OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

    std::cout << res;
}
```





# Plusieurs branches sur origin

## *Tirer parti des capacités de git à gérer les branches*

La manière de travailler précédente ("à la SVN") possède **plusieurs inconvénients** qui résultent de ce constat:

Puisqu'il semble évident qu'on s'interdise de partager dans "master sur origin" des développements non terminés et non validés (par une batterie de tests), le travail "en cours" du développeur reste donc sur sa machine.

### Conséquences négatives:

- Le développeur ne peut pas facilement changer de machine pour travailler (fichier .zip à transférer d'une machine à l'autre),
- Personne n'est au courant de ce qu'il est en train de coder dans son coin,
- S'il quitte le projet (fin de thèse p.expl.) et qu'il ne prend pas le temps de merger (ça arrive), le projet se retrouve au mieux avec le zip plus haut, au pire avec rien du tout!

Le workflow présenté ci-après permet de résoudre tous ces problèmes.



# Plusieurs branches sur origin

L'idée est simplement ***d'envoyer les branches de développement sur origin***

```
git clone git@gitlab.uliege.be:R.Boman/code.git
```

```
git checkout -b boman
```

```
vi README.md
```

```
git add README.md
```

```
git commit -m "improved README"
```

```
git push
```

```
fatal: The current branch boman has no upstream branch.
```

```
To push the current branch and set the remote as upstream, use
```

```
git push --set-upstream origin boman
```

on crée une branche  
boman

(rappel: on utilise -b  
uniquement la 1ère  
fois où on crée la  
branche)

raccourci:

--set-upstream => -u

La branche boman n'existe pas sur  
origin

git ne crée pas par défaut une branche  
de même nom sur origin



# Plusieurs branches sur origin

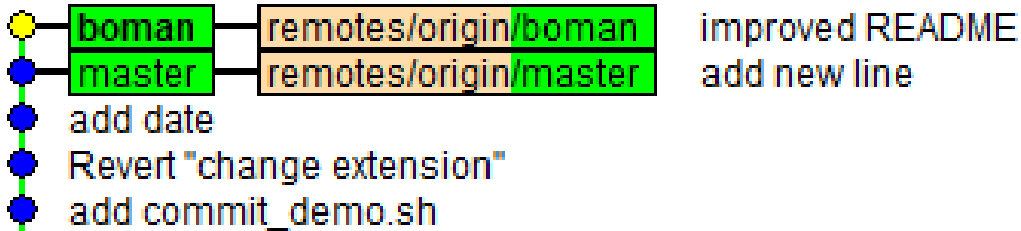
## Envoyer sa branche de développement sur origin

```
git push -u origin boman
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 347 bytes | 173.00 KiB/s,
Total 3 (delta 1), reused 0 (delta 0)
[...]
To gitlab.uliege.be:R.Boman/code.git
* [new branch]      boman -> boman
```

envoie la branche courante sur origin, la nomme "boman sur origin"

Ajoute une branche origin/boman au repository local qui sera mis à jour avec la commande `git fetch`

"boman" devient une "tracking branch" de origin/boman (grâce à -u)





# Plusieurs branches sur origin

...Zut, j'ai oublié le -u! (erreur courante)

```
git push origin boman
Enumerating objects: 5, done.
[... ca marche ...]
```

Si on oublie le -u, ça fonctionne, mais "boman" ne sera pas une *"tracking branch"*

Conséquence:  
git pull sans argument ne marchera pas!

```
git pull
```

There is no tracking information for the current branch.  
Please specify which branch you want to merge with.  
See git-pull(1) for details.

```
git pull <remote> <branch>
```

soit on fait chaque fois:  
git pull/push origin boman

If you wish to set tracking information for this branch you can do so with:

```
git branch --set-upstream-to=origin/<branch> boman
```

soit on corrige  
une fois pour  
toutes

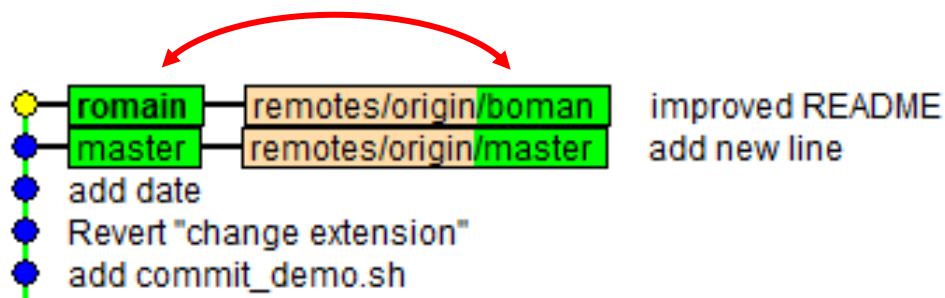


# Plusieurs branches sur origin

ATTENTION: Les branches locales peuvent avoir des noms différents des branches sur origin (très déconseillé!)

```
git checkout -b romain  
git push -u origin boman
```

on envoie romain vers  
"boman sur origin"



Ceci montre encore une fois la très grande flexibilité de git  
(qui nuit bien souvent à son apprentissage).



# Plusieurs branches sur origin

## Lister les branches en ligne de commande

```
git branch -avv
```

master	8cc4b86	[origin/master]	add new line
* <b>romain</b>	8abbbe5	[origin/boman]	improved README
remotes/origin/HEAD	->	origin/master	
remotes/origin/boman	8abbbe5		improved README
remotes/origin/master	8cc4b86		add new line

tracking info

branche  
locale  
courante

désigne la branche  
principale du projet  
(peut être autre  
chose que master)

Cette commande ne donne aucune info sur les branches sur `origin` qui peuvent avoir évolué depuis le dernier fetch!

Pour 2 branches sur `origin`, on a donc ici **6** branches à gérer (en plus de la working copy et du staging index). Donc **8 versions potentiellement différentes du code!**



# Plusieurs branches sur origin

## Retour à "git clone"

Travailler sur sa branche à partir d'un clone "vierge"

```
git clone git@gitlab.uliege.be:R.Boman/code.git
```

```
git branch -avv
```

```
* master          8cc4b86 [origin/master] add new line
remotes/origin/HEAD -> origin/master
remotes/origin/boman 8abbbe5 improved README
remotes/origin/master 8cc4b86 add new line
```

git ne crée pas de  
branche locale boman.

Seul master est créée

```
git checkout boman
```

```
Switched to a new branch 'boman'  
Branch 'boman' set up to track remote branch 'boman' from 'origin'.
```

ATTENTION: pas -b !

git branch boman origin/boman  
est implicite:  
git nous a épargné la création explicite de la  
branche et de son info de tracking.

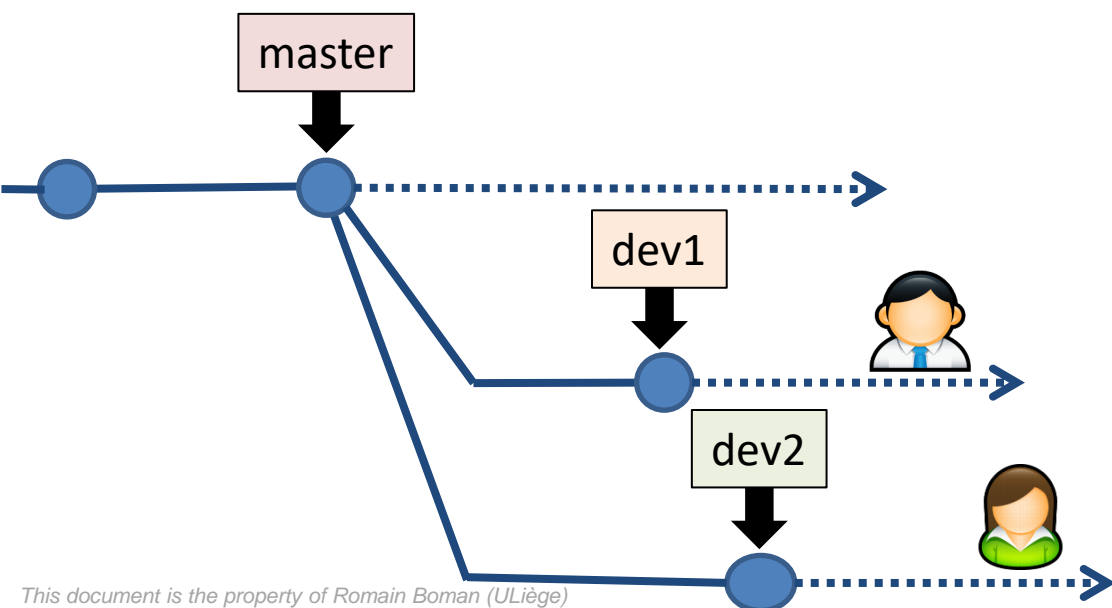


# Plusieurs branches sur origin

## ***WORKFLOW "chacun dans sa/ses branche(s)"***

Principe 1: la branche master est toujours stable (master pointe vers une version qui compile sans erreur et a passé une batterie de tests).

Principe 2: les développeurs se créent une/des branche(s) personnelles liées à leur travail. Ils créent des commits locaux au fur et à mesure des développements. Ils envoient leurs commits régulièrement sur GitLab.



master ne recevra que des "merges" des autres branches.

Personne ne travaille directement dans master.



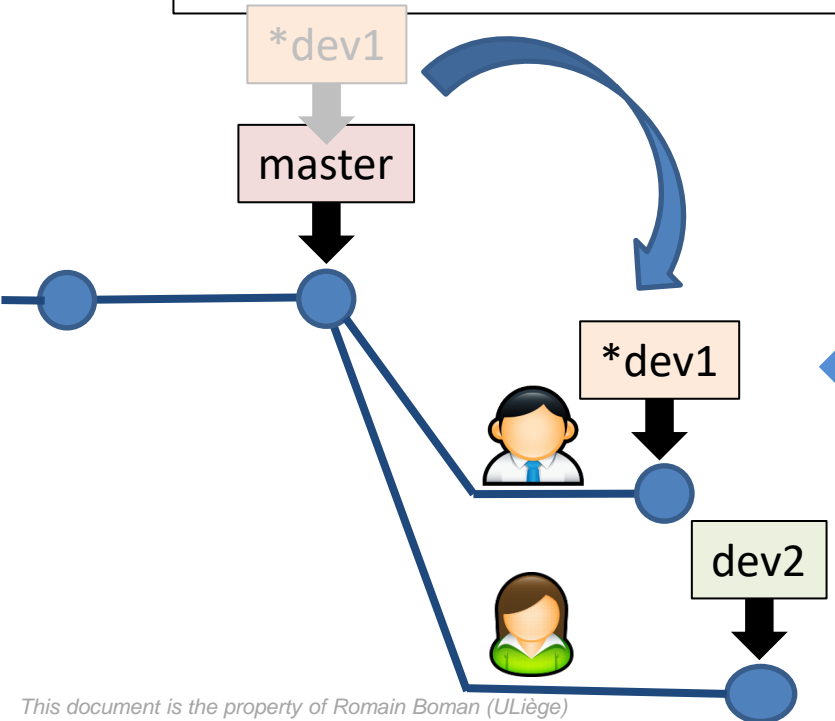


# Plusieurs branches sur origin



```
git clone git@gitlab.uliege.be:R.Boman/code.git
git checkout -b dev1
git push -u origin dev1
[travail]
git add .
git commit -m "fix bug XXX"
git push
[...]
```

Intérêt: les versions en cours de développement sont sauvegardées (backup), visibles (interactions avec les autres) et facilement récupérables d'une machine à l'autre (changement de PC).




la branche locale courante est désignée par une \*

Note: on imagine qu'un 2e développeur fait la même chose en parallèle

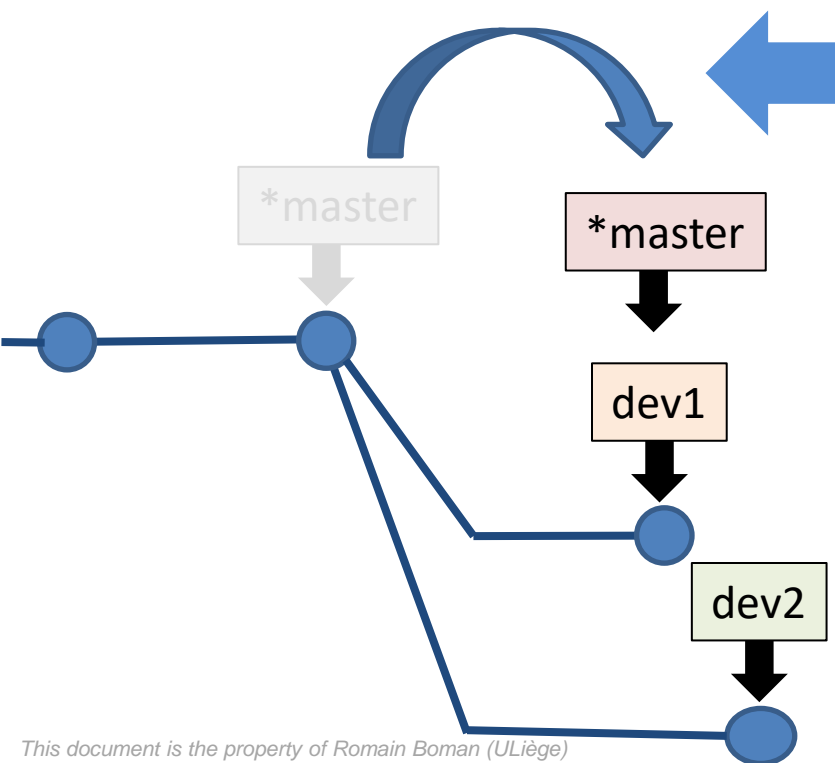


# Plusieurs branches sur origin


Principe 3: si le développeur est à jour avec master, il peut passer la batterie de tests et merger sa version dans master (équivalent à un "svn commit "  ):



```
[batterie de tests OK]  
git checkout master  
git merge dev1  
git push
```



Ce type de merge est un "*fast-forward merge*". Il consiste simplement à déplacer le pointeur master vers dev1.

Contrairement à SVN, il n'y a pas de nouveau commit dans ce cas. 

En particulier, quelqu'un qui aurait déjà intégré les commits de dev1 dans sa propre branche est déjà à jour avec le nouveau master!

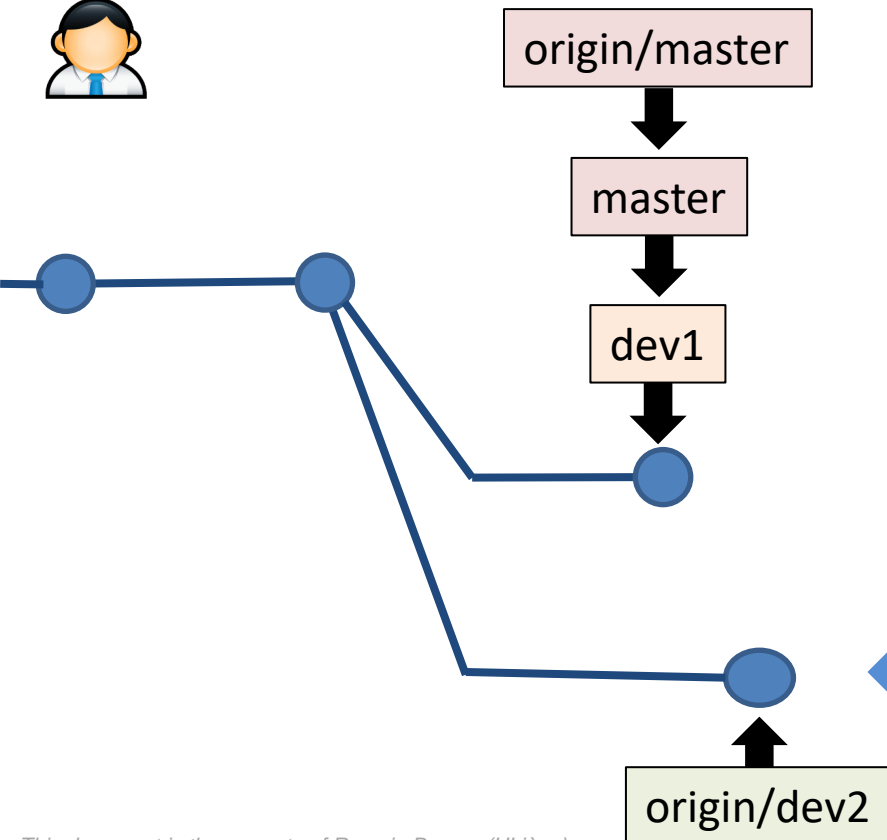


# Plusieurs branches sur origin

## PARENTHÈSE TRES IMPORTANTE (1/2):

Arrêtons-nous un moment pour nous demander quel est l'état du repository local des 2 développeurs à ce moment précis (après un `git fetch` de chaque côté).

### REPO LOCAL DEV. 1



### Source d'incompréhension courante #1:

Il n'y a pas de branche `dev2` chez développeur 1! (il n'a jamais fait un `git checkout dev2`).

Par contre, le dernier commit de développeur 2 est accessible via la branche `origin/dev2` qui a été récupérée par le `git fetch`.

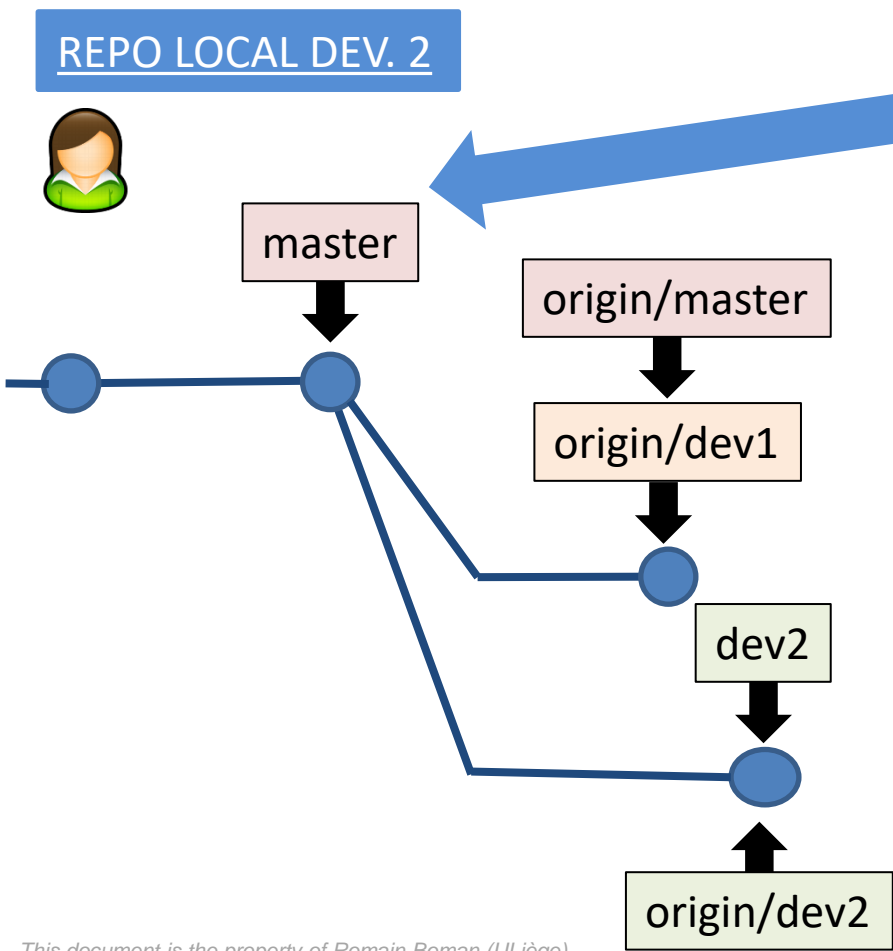
*Remarque:* si le développeur 1 voulait "voir" cette branche localement (ce qui n'est généralement pas nécessaire), il devrait faire `git checkout dev2` qui associerait une branche locale `dev2` à `origin/dev2`



# Plusieurs branches sur origin

**PARENTHESE TRES IMPORTANTE (2/2):**

Arrêtons-nous un moment pour nous demander quel est l'état du repository local des 2 développeurs à ce moment précis (après un `git fetch` de chaque côté).



Source d'incompréhension courante #2:

master du développeur 2 désigne toujours le dernier commit de la branche master au moment où elle à fait son `git clone`!


En effet, elle n'est jamais retournée sur master pour faire un `git pull`.

Par contre, `origin/master` désigne bien le même commit que "master sur origin" car elle a fait un `git fetch`.

De plus, il n'y a pas de branche dev1 chez développeur 2.



# Plusieurs branches sur origin

Principe 4: si le développeur n'est pas à jour avec master, il peut mettre à jour sa branche avec le contenu de master (équivalent à un `svn update` )

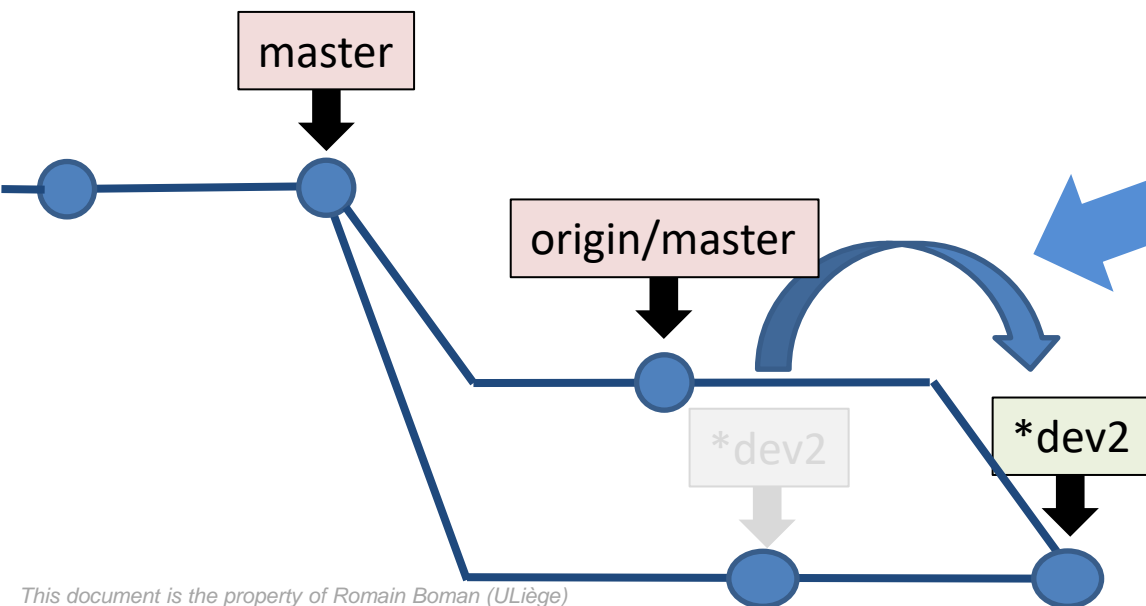


```
git checkout dev2
git fetch
git merge origin/master
[résolution des éventuels conflits]
git push
```

Il s'agit cette fois d'un "*recursive merge*" qui crée un nouveau commit dans la branche dev2.

master n'est pas impacté.

Le propriétaire de dev2 se retrouve alors dans le cas précédent (il peut passer ensuite la batterie et envisager un merge dans master).





# Plusieurs branches sur origin

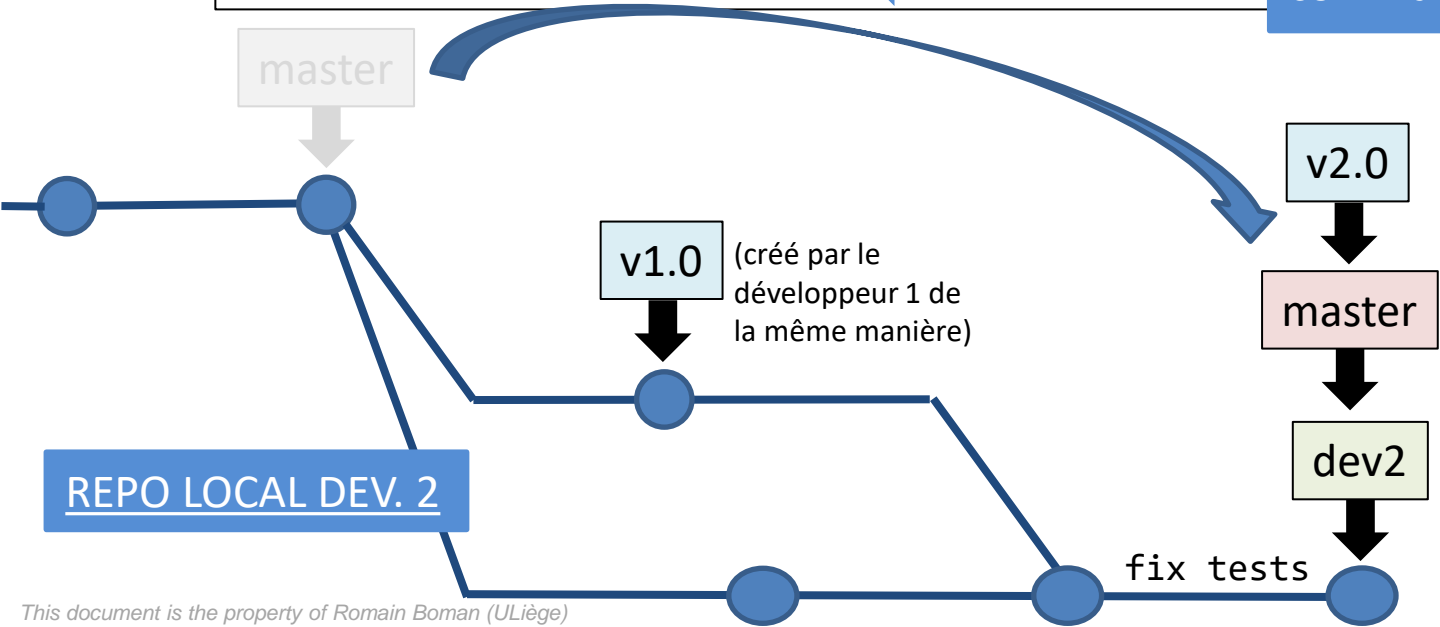
Principe 5: les versions stables sont taguées (pour éviter de les confondre avec des commits intermédiaires qui n'ont pas passé la batterie de test).



```
git commit -m "fix tests"  
[batterie de test OK]  
git checkout master  
git merge dev2  
git push  
git tag v2.0  
git push origin v2.0
```

met à jour master  
et envoie sur  
origin

git tag crée un tag local.  
Le tag local doit être ensuite  
envoyé sur origin avec une  
commande push.





# Plusieurs branches sur origin

## **Nettoyage (prudence!)**

Supprimer un tag sur origin

```
git push origin :v1.0
```

même commande que pour le créer mais avec un ":" précédant le nom du tag

Supprimer une branche sur origin

```
git push origin :dev1
```

tag et branches sont similaires. Ce sont des références à un commit particulier

Supprimer remotes/origin/dev1

```
git branch -rd origin/dev1
```

"dev1 sur origin" et "origin/dev1" sont 2 branches distinctes!

Lister les tags et branches sur origin:

```
git ls-remote --refs
```

Supprimer toutes les branches origin/xxx dont la branche sur origin n'existe plus:

```
git fetch origin --prune
```



# Plusieurs branches sur origin

## ***ATTENTION: "Supprimer" un commit envoyé sur origin?***

C'est (partiellement) possible... mais hautement déconseillé!

### Pourquoi?

- Le commit est sur un repository partagé. Il est donc peut-être déjà utilisé par quelqu'un autre (peut-être vous-même, sur une autre machine)
- Comme pour la correction de commits locaux, la "suppression" va consister à déplacer la référence "branche sur origin" vers l'arrière, ou réécrire (par rebase) l'historique. Le commit envoyé ne sera donc pas immédiatement supprimé!

### Conséquences:

- A faire donc uniquement pour corriger une branche personnelle ou une catastrophe qui vient de se produire (envoi d'un fichier énorme non voulu sur origin tel un .zip ou des résultats de calcul, envoi d'un fichier de mots de passe, etc.)
- Dans tous les cas, avertir les autres développeurs!

### Prévention:

- Toujours vérifier ce qui va être commité (`git status`)
- Ne pas faire des commit/push à la mitrailleuse!





# Plusieurs branches sur origin

## *"Supprimer" une grosse erreur envoyée sur origin (1/3)*

Exemple: on envoie une vidéo sur GitLab

```
du -sh .git
92K  .
git add movie.mp4
git commit -m "add movie.mp4"
git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 6 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 21.93 MiB | 7.75 MiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
du -sh .git
23M  .git
```

La vidéo de 21.93Mo est dans l'historique!

Evidemment, effacer la vidéo et refaire un commit ne va rien arranger: git va garder précieusement trace de notre erreur.

A partir de ce moment, tous les utilisateurs qui font un clone vont récupérer la vidéo (c'est à dire 23Mo au lieu des 92Ko du repository avant l'erreur)



# Plusieurs branches sur origin

## ***"Supprimer" une grosse erreur envoyée sur origin (2/3)***

La solution consiste à revenir à faire reculer le pointeur de branche vers un ancien commit sur le repository local, comme vu précédemment

```
git reset --hard HEAD^
```

... et forcer le push (un push classique ne va pas être accepté)

```
git push --force
```

A cet instant, git ignorera le commit lors d'un `git clone` et la vidéo ne sera plus téléchargée

...mais elle existe toujours car il est référencé dans l'onglet "Activity"

20 Aug, 2019 1 commit



add movie.mp4

Romain Boman authored 3 hours ago

f303e0dc



19 Aug, 2019 2 commits



add doc for solver.py

Romain Boman authored 1 day ago

3717391f



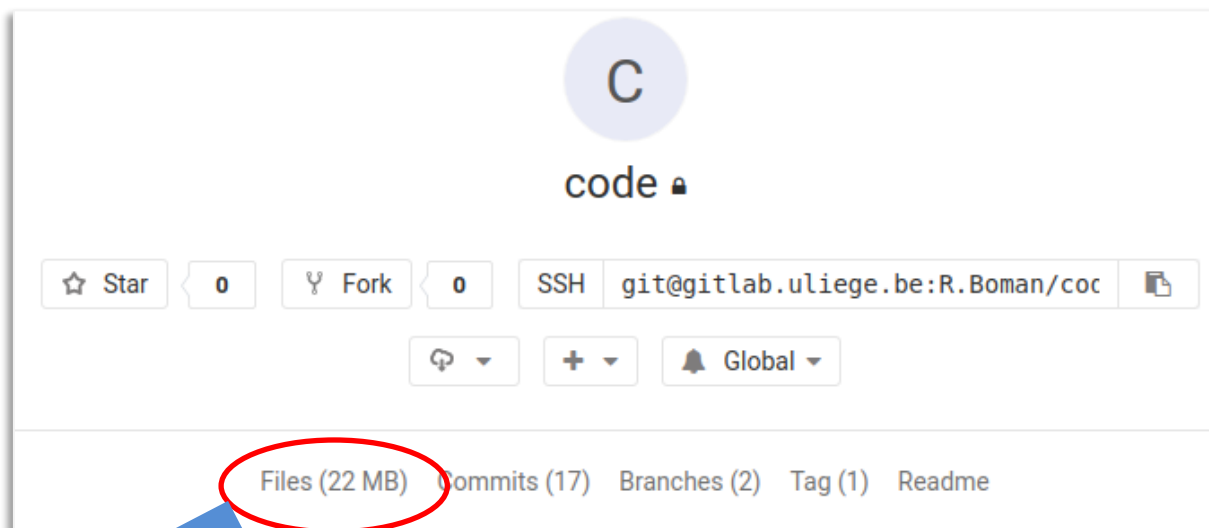
voir p expl:

<https://gitlab.com/gitlab-org/gitlab-ce/issues/30358>



# Plusieurs branches sur origin

***"Supprimer" une grosse erreur envoyée sur origin (3/3)***



La video est toujours dans les fichiers gérés par GitLab !

Les outils de maintenance du repository ("housekeeping") ne la supprime pas (du moins, pas avant plusieurs mois!)

Conclusion: même s'il existe des moyens pour limiter les dégats, envoyer un gros fichier non voulu sur GitLab via un push incontrôlé est la plus grosse erreur qu'on puisse imaginer.



# Plusieurs branches sur origin

## Utilisation avancée: Comment nettoyer le repository local?

Supprime toutes les références aux commits perdus dans le reflog (en particulier celui où on a ajouté la vidéo).

```
git reflog expire --expire=now --expire-unreachable=now --all
```

```
git prune --verbose --expire=now
```

élague (prune) les branches mortes qui ne sont plus référencées

```
git gc
```

démarre le garbage collector (optimise le repo local)

```
du -sh .git
```

80K

la vidéo a disparu.



# Plusieurs branches sur origin

## **Cas pratique #1: origin a changé de nom ou d'emplacement**

Imaginons que le repository soit renommé (ou déplacé sur un autre serveur, à une autre adresse)

Il suffit de changer l'adresse d'origin par la commande:

```
git remote set-url origin git@gitlab.uliege.be:R.Boman/code2.git
```



Commande équivalente SVN : `svn relocate`



# Plusieurs branches sur origin

## **Cas pratique #2: Comment ne pas perdre mes modifications en changeant de branche? (git stash)**

```
vi README.md
```

J'ai modifié le code mais je me rends compte trop tard que je suis sur master. Je dois revenir sur la branche boman

```
git checkout boman
```

```
error: Your local changes to the following files would be overwritten by checkout:  
    README.md
```

```
Please commit your changes or stash them before you switch branches.
```

```
Aborting
```

```
git stash
```

```
git checkout boman
```

```
git stash pop
```

```
Auto-merging README.md
```

```
CONFLICT (content): Merge conflict in README.md
```

```
[résoudre le conflit]
```

"stash" sauvegarde les modifications dans une pile utilisable par la suite. La commande "stash pop" réapplique les modifs sauvegardées.

Si j'avais copié README.md sur mon bureau puis écrasé la version après checkout, j'aurais effacé les évolutions de README.md sur master!

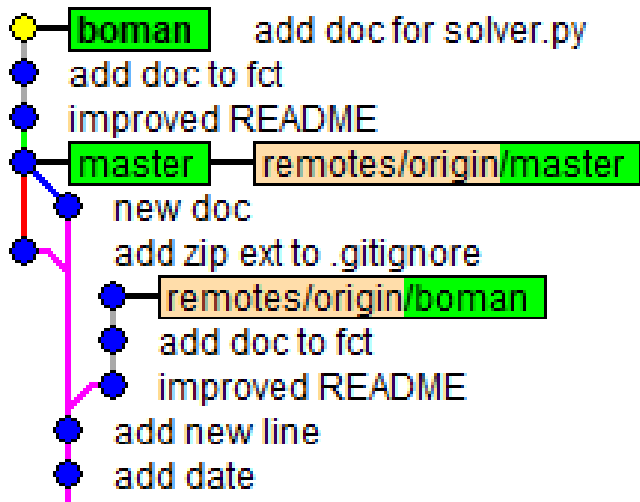
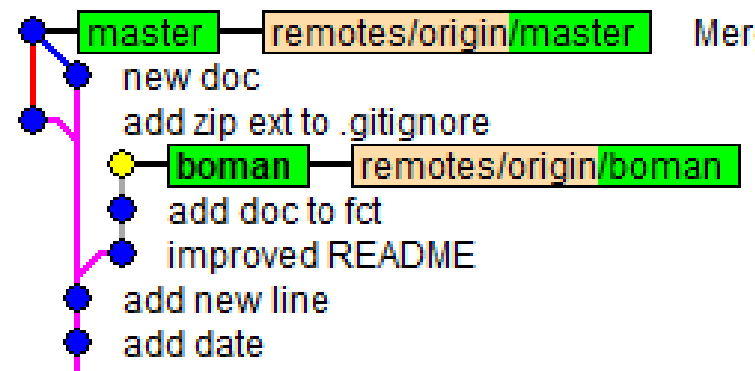


# Plusieurs branches sur origin

## Cas pratique 3: Peut-on réécrire sa branche avec "git rebase"?

Techniquement, oui, mais c'est (très) déconseillé!

```
git checkout boman
git fetch
git rebase origin/master
```



"git rebase" rejoue les commits sur master. Les anciens commits sont toujours là!

Un simple "git push" n'est plus possible car il ne s'agit plus d'un *fast-forward merge*!

On retrouve tous les problèmes liés au cas de la suppression de commits!



# Plusieurs branches sur origin

## ***Conclusions***

### **Quelles sont les règles supplémentaires à retenir par rapport à un travail local (partie 1)?**

- On ne travaille jamais sur master (master va vite se désynchroniser de origin/master!)
- Ne pas systématiquement faire push après commit.
  - Vérifier ses commits et prendre le temps de les nettoyer tant qu'ils ne sont pas sur origin.
- A moins de savoir ce qu'on fait, ne pas faire de "push --force", c'est-à-dire:
  - Ne pas supprimer des commits sur origin.
  - Ne pas faire de rebase d'une branche déjà sur origin (utiliser git merge).
- En cas d'erreur sur origin, prévenir les autres développeurs.



# Utiliser git

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int tmax)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            c[i*n+j] = a[i*n+j] + b[i*n+j];
}
```

## PARTIE 3

### En pratique...

```
double tstop = omp_get_wtime();
double cpu = tstop-tstart;

OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

std::cout << res;
```



# En pratique

## *Que manque-t-il pour travailler encore plus efficacement?*

- Un **éditeur de texte** qui "intègre" git :
  - je propose ici Visual Studio Code
  - il en existe plein d'autres...
- Une procédure d'évolution "sécurisée" du code et la définition de **versions stables**: Merge Requests
- Un suivi des **bugs** : Issues
- Un système de **tests** automatiques : Intégration Continue (CI)
- Un système de **notifications** de l'activité sur le repository et un moyen de **discussion** entre développeurs : Slack

# Utiliser git

PARTIE 3  
En pratique...

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int max)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        #pragma omp parallel for num_threads(nbt)
```

## Utiliser git à travers un IDE

Visual Studio Code

```
double tstop = omp_get_wtime();
double cpu = tstop-tstart;

OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

std::cout << res;
```



# Visual Studio Code

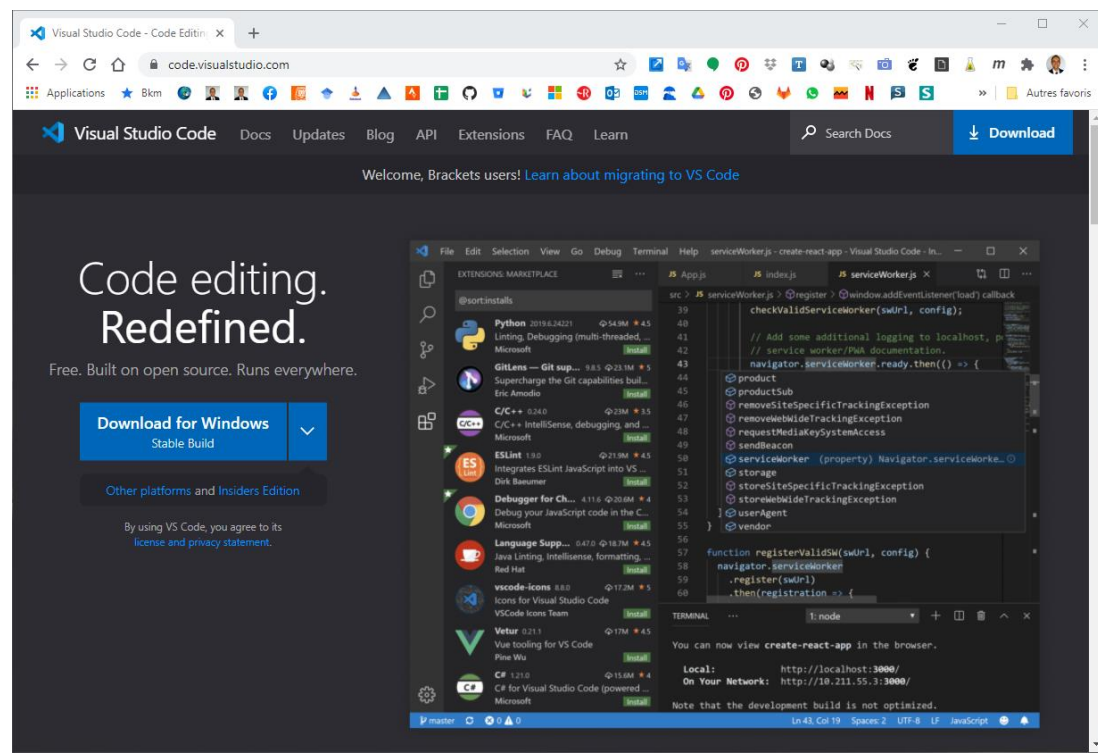
Visual Studio Code (ou VSCode)...

- est développé par Microsoft
- ne doit pas être confondu avec Visual Studio
- est un éditeur léger
- intègre git dans son interface
- facilite les opérations telles que diff/merge
- est multi-plateforme (Windows, Linux, macOS)

<https://code.visualstudio.com/>

Il est nécessaire de comprendre et maîtriser un minimum les commandes git pour utiliser VSCode.

Surtout ne pas faire du git "à l'instinct" à travers cette interface!



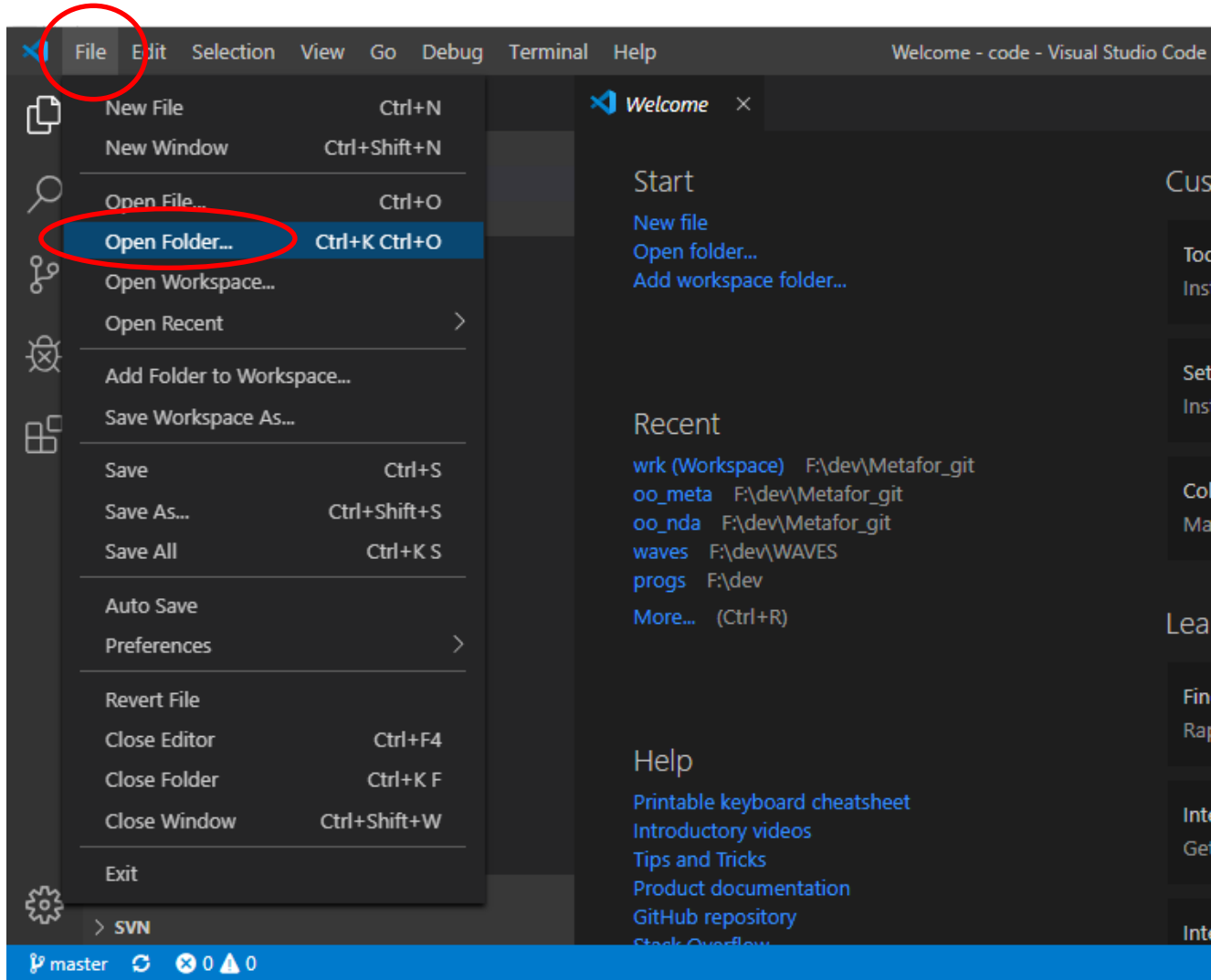


# Visual Studio Code

## Ouvrir un répertoire

Il n'y a pas de "projet" dans VSCode. On travaille dans un répertoire.

=> choisir un répertoire de travail déjà cloné.





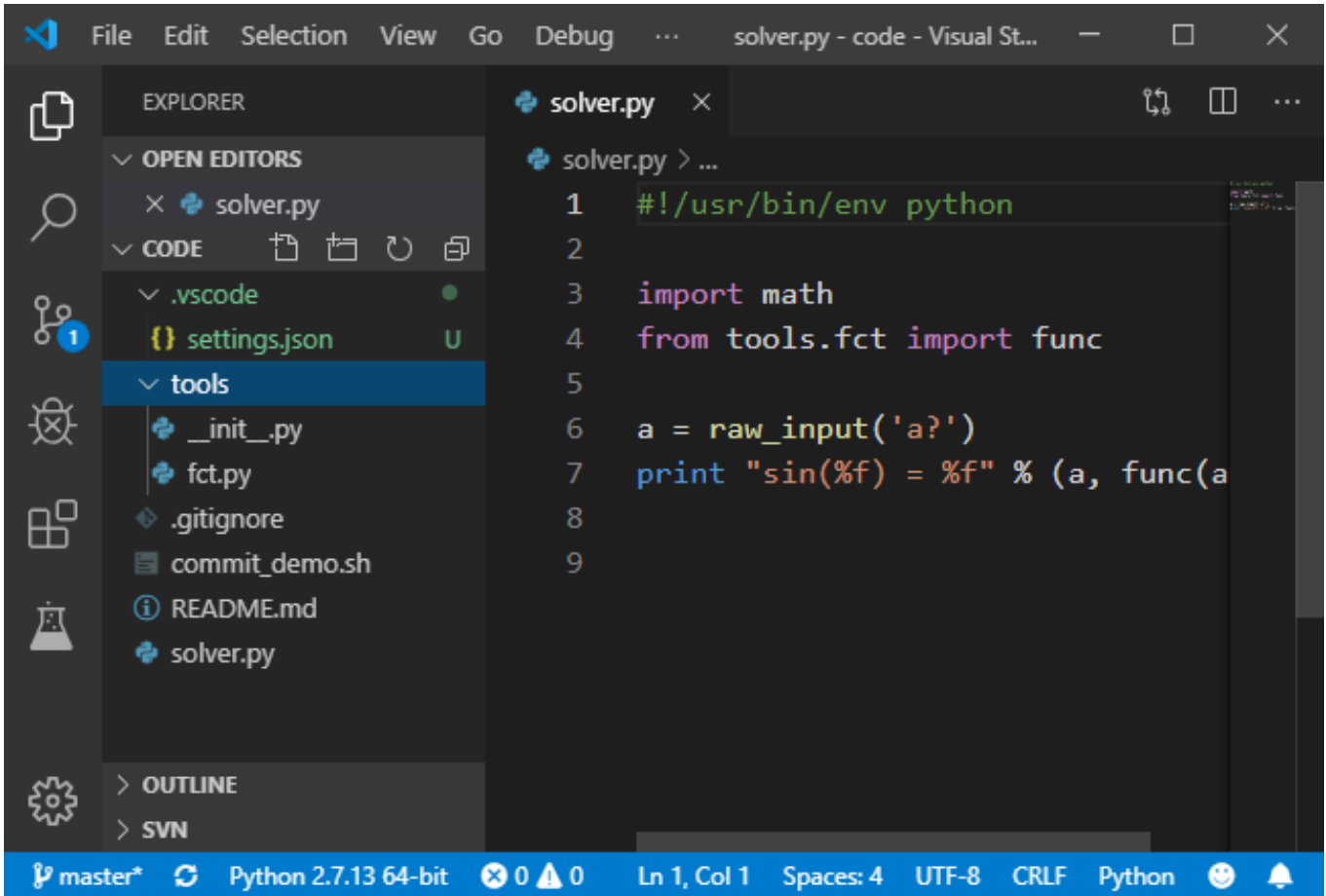
# Visual Studio Code

Il est possible d'installer des **extensions** pour Python, C++, Fortran, CMake, Matlab, etc.

On s'intéresse ici aux fonctionnalités git qui sont disponibles **par défaut**

Fonctions git

Extensions





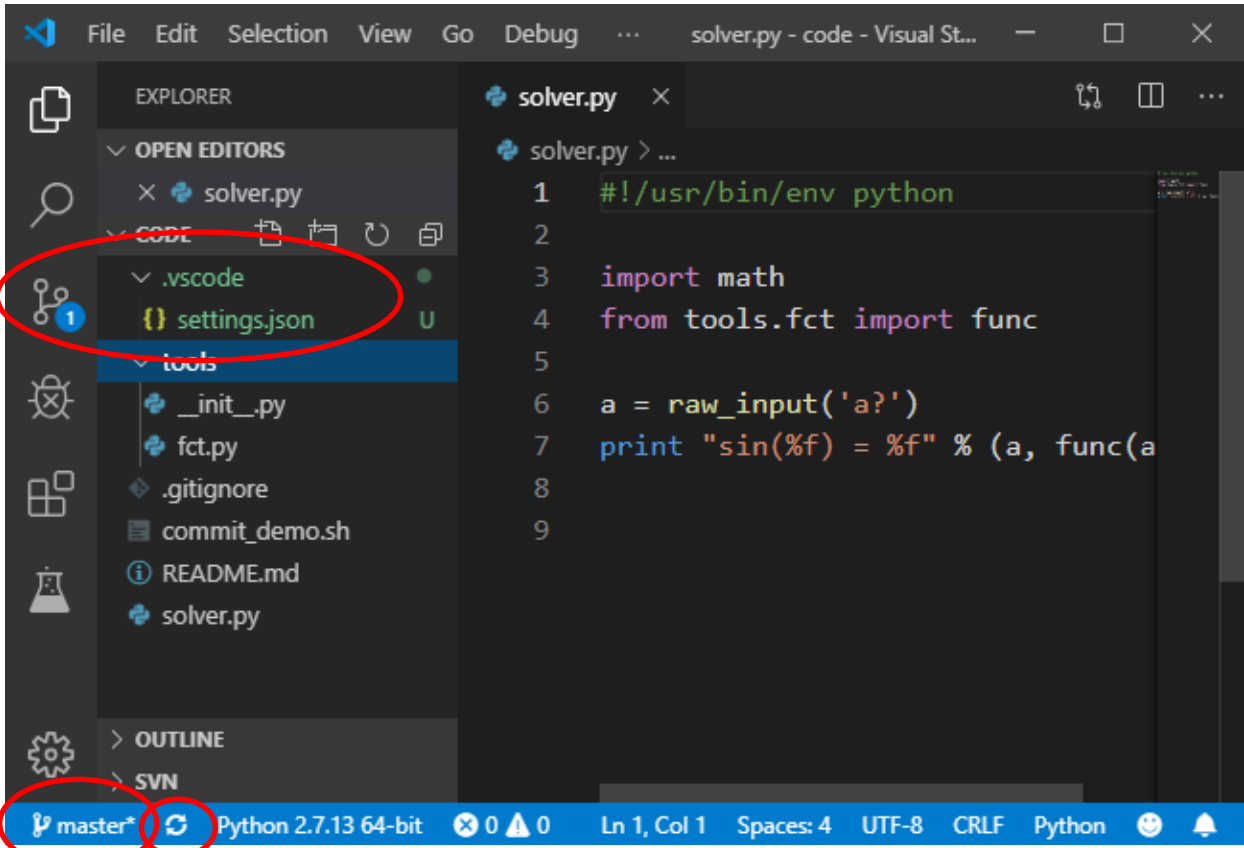
# Visual Studio Code

## L'interface "git"

VSCode stocke sa configuration dans un dossier `.vscode/`

Il a créé un nouveau fichier `settings.json`

Permet de changer de branche (git checkout) ou créer une branche (git branch)  
La branche courante est master (\* indique "modifié").



effectue un "git pull" – VSCode fait également des "git fetch" réguliers pour montrer l'état du repo local.





# Visual Studio Code

## ***Exemple d'utilisation***

Contexte: On veut ajouter le dossier `.vscode/` à `.gitignore` pour que son contenu soit ignoré par git (il contient la configuration propre à l'utilisateur).

Si on était en ligne de commande, on ferait, par exemple:

```
vi .gitignore  
[ajout d'une ligne: .vscode/ ]  
  
git diff .gitignore  
  
git add .gitignore  
  
git status  
  
git commit -m "add .vscode to ignore list"  
  
git push
```





# Visual Studio Code

Equivalent de... `vi .gitignore`

sauvegarde

EXPLORER

- 0 1 UNSAVED
- .gitignore
- CODE
  - .vscode
    - settings.json
  - tools
    - \_\_init\_\_.py
    - fct.py
  - .gitignore
  - commit\_demo.sh
  - README.md
  - solver.py
- OUTLINE
- SVN

.gitignore

```
1 *.pyc
2 *~
3 *.tmp
4 *.zip
5 *.bak
6 .vscode/
```

montre l'endroit "modifié" du fichier (git diff)

ligne ajoutée

master Python 2.7.13 64-bit 0 0 Ln 6, Col 9 Spaces: 4 UTF-8 CRLF Ignore

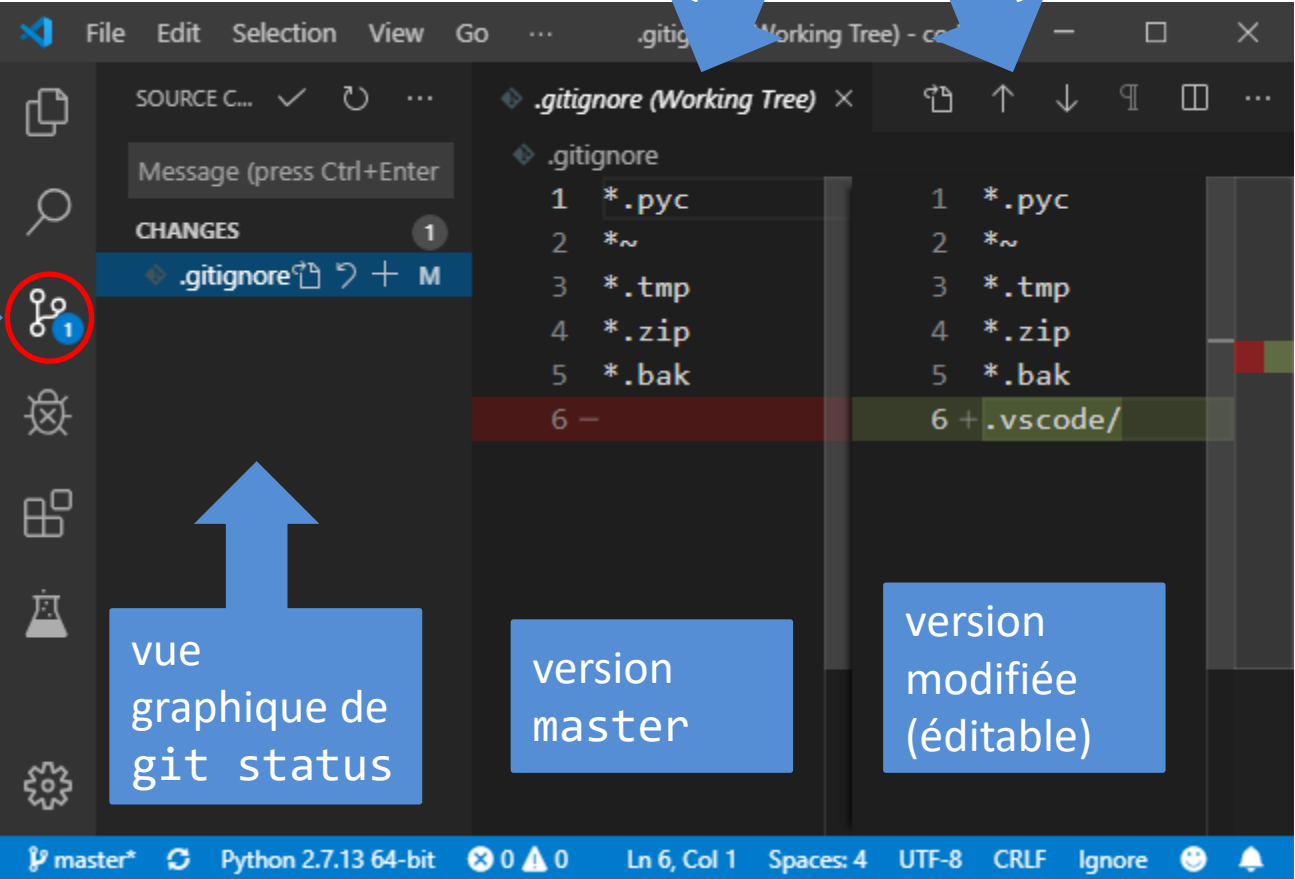


# Visual Studio Code

Equivalent de... `git diff .gitignore`

`git diff`  
"à la Araxis Merge"

Cliquer sur  
cette onglet et  
sélectionner le  
fichier  
.gitignore

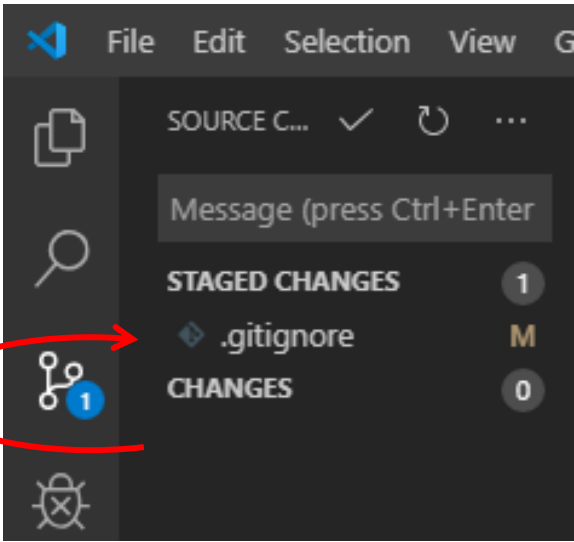
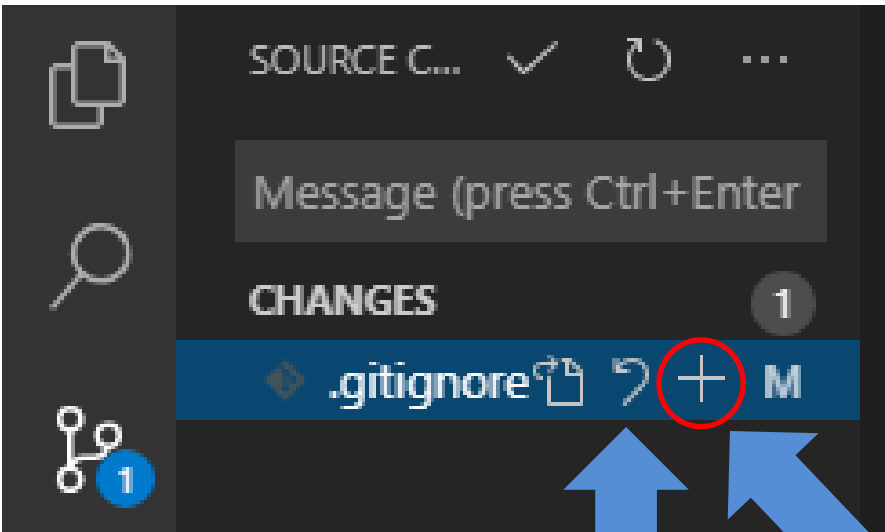




# Visual Studio Code

Equivalent de...

```
git add .gitignore
```



git add .gitignore  
(ajoute les mods à l'index)

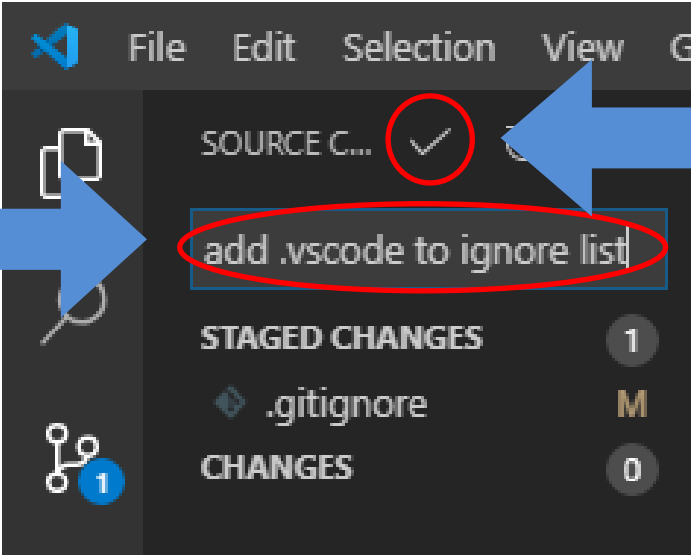
git checkout -- .gitignore  
(supprime les modifications)



# Visual Studio Code

Equivalent de... `git commit -m "add .vscode to ignore list"`

étape1:  
écrire le message ici



étape2:  
git commit



L'état du repository local est visible ici:  
on est 1 commit "en avance" sur le  
master d'origine et 0 commits "en retard"

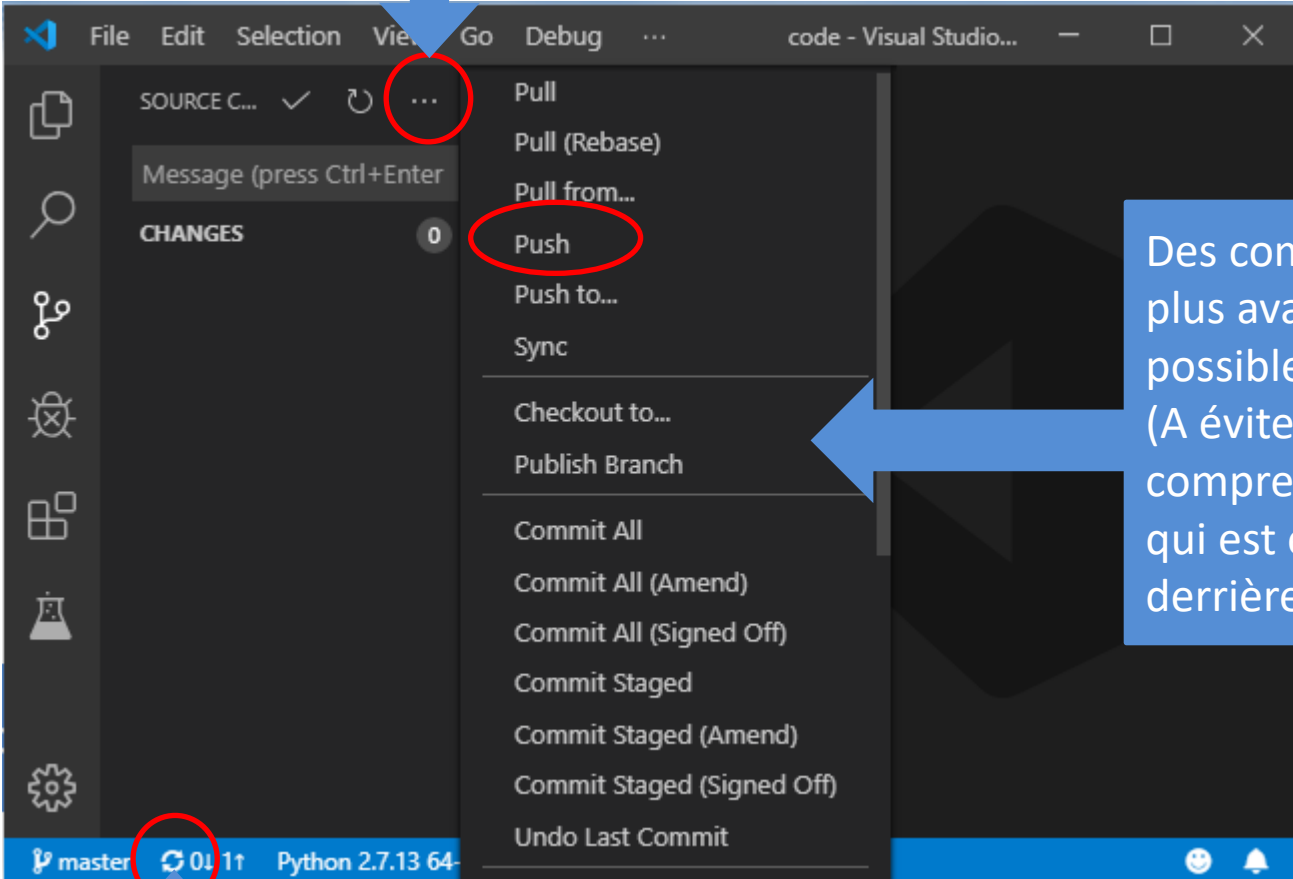


# Visual Studio Code

Equivalent de...

`git push`

Utiliser ce menu...



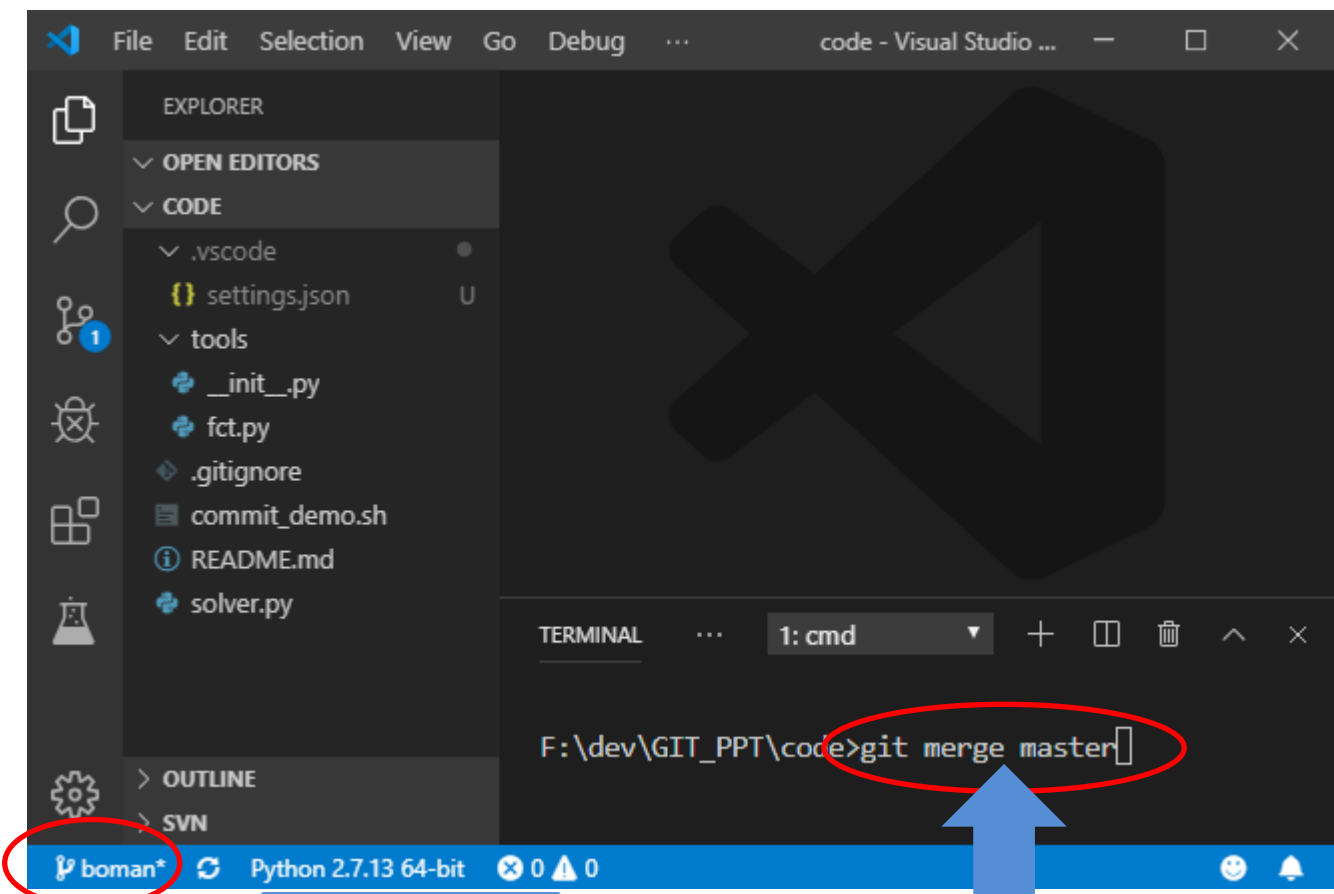
Des commandes plus avancées sont possibles.  
(A éviter si on ne comprend pas ce qui est exécuté derrière!)

..ou cliquer ici

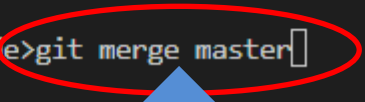


# Visual Studio Code

Les opérations de fusion (merge) sont également très simples:



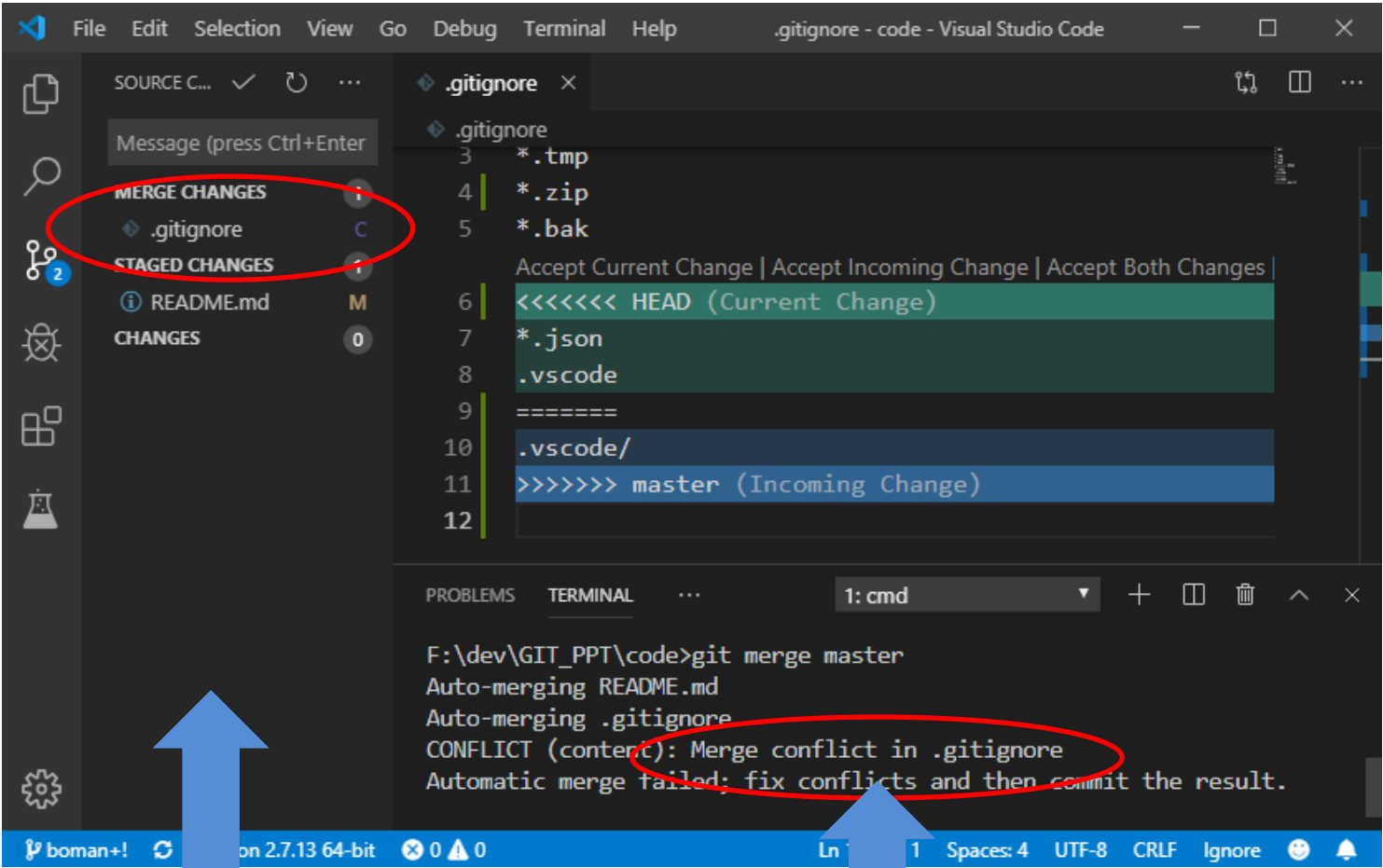
Changer de  
branche en  
cliquant ici



Dans le terminal intégré (ou un git bash  
externe), exécuter "git merge master"



# Visual Studio Code



Nouvelle section ici: les fichiers en conflit ("C")

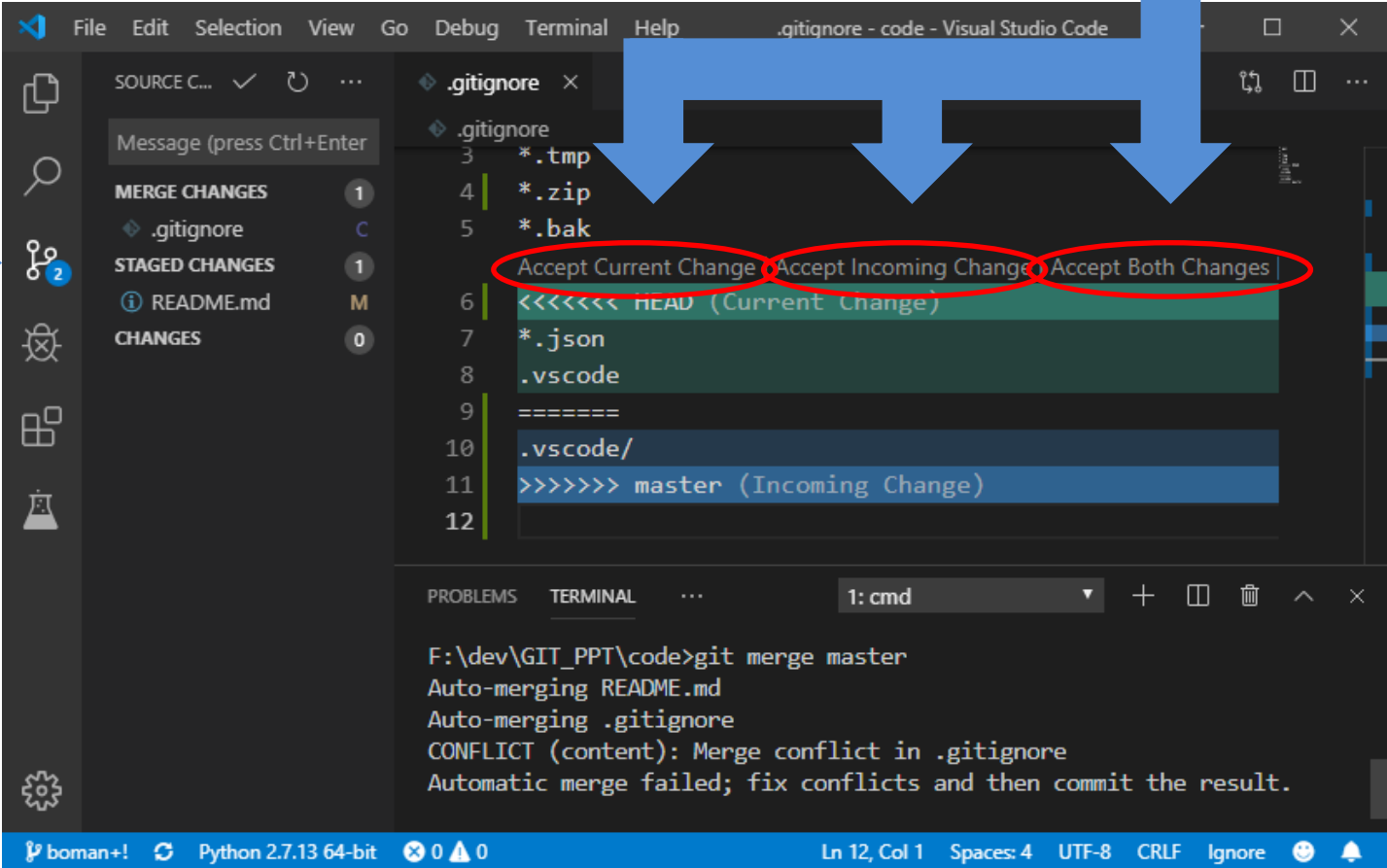
La fusion a produit un conflit.



# Visual Studio Code

Les modifications de master qui n'ont pas posé de problème ont été "ajoutées" (ici README.md). On peut les visualiser les diffs en cliquant dessus.

Aide à la résolution des conflits: 3 boutons par démarrer le travail

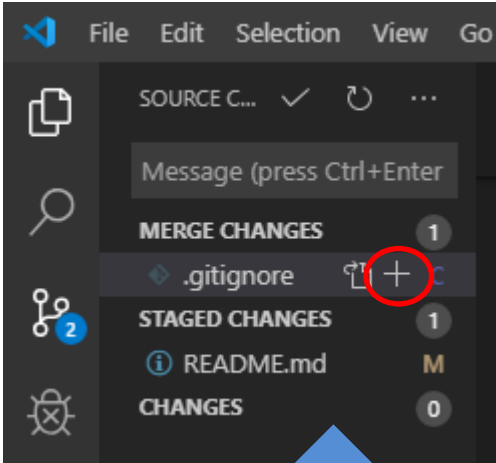




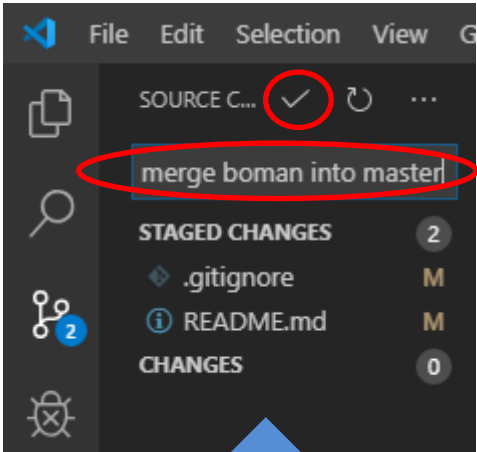


# Visual Studio Code

Une fois le conflit résolu:



Faire "git add" pour marquer la résolution (bouton "+")



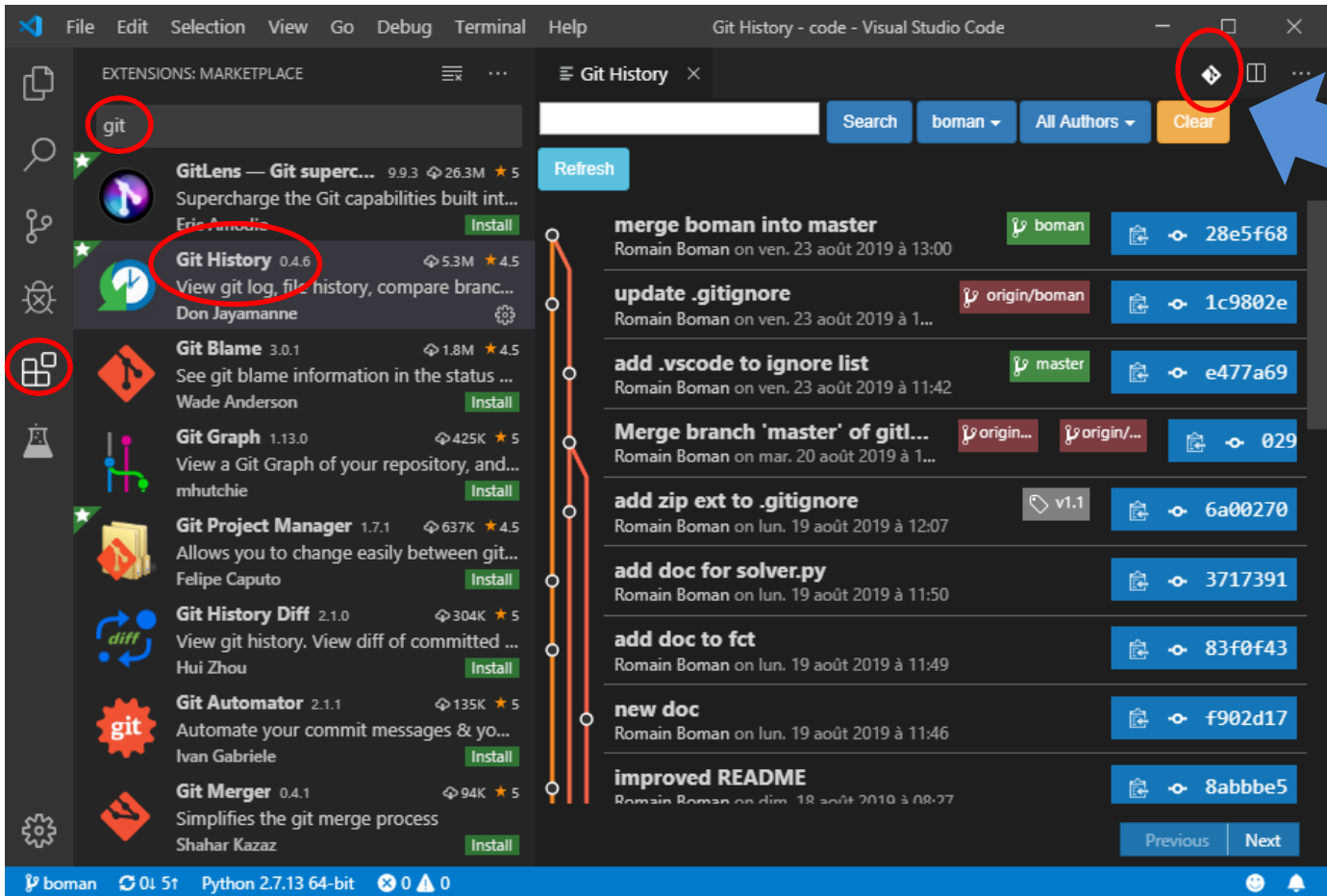
Commiter le résultat

Vérifier que la section "MERGE CHANGES" (les conflits) et CHANGES (des modifications qui auraient été faites après le merge à d'autres fichiers) sont vides.



# Visual Studio Code

Des dizaines d'**extensions** existent pour aller plus loin:



Par exemple, *Git History* ajoute ce bouton pour voir l'arbre des commits un peu comme le fait "gitk --all"

# Utiliser git

PARTIE 3  
En pratique...

Merge requests, issues, CI pipelines,...  
...pour travailler comme un pro.

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int max)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        #pragma omp parallel for num_threads(nbt)
        for (int j=0; j<n; j++)
        {
            c[i*n+j] = a[i*n+j] + b[i*n+j];
        }
    }
}

double tstop = omp_get_wtime();
double cpu = tstop-tstart;

OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

std::cout << res;
```



# Merge requests

Un "***merge request***" (ou "*pull request*" sur github) est un **temps d'arrêt et de réflexion** lors d'une opération "`git merge`".

L'idée est de demander (*to request*) l'autorisation d'effectuer un *merge*, généralement vers une branche sensible (`master`).

L'origine du merge request peut être n'importe quelle branche de développement, y compris une branche d'un "fork" (copie du repository).



Dans une procédure SVN à 1 branche, le "merge request" correspond à un **"commit"**, c'est-à-dire à la définition d'une **nouvelle version stable** du code qui pourra être référencée par la suite (en termes GitLab, on parle de *Release*).



# Merge requests

## *Quel sont les buts?*

- **Protection de master**

La version master devrait être toujours fiable. En particulier elle doit passer avec succès la batterie de tests. Il faut donc s'interdire de faire un "git push origin master" sans précaution. GitLab interdit tout push vers master sans passer par un Merge Request aux membres du projet de type "Développeurs".

- **Communication**

C'est le moment de mettre à plat ce qui a été fait dans sa branche de développement et de le décrire en long et en large aux autres (et pour soi-même).

- **Review**

On donne l'occasion aux autres développeurs de regarder les mods, les commenter et/ou demander des corrections.

- **Nettoyage**

C'est l'occasion de nettoyer son code (std::cout qui traînent, tabulations, doc, etc.)



# Merge requests

## Protection de la branche master

code

Project

Repository

Issues0

Merge Requests1

CI / CD

Security & Compliance

Operations

Wiki

Snippets

Settings

General

Members

Integrations

Repository

CI / CD

Operations

Audit Events

Protected Branches

Keep stable branches secure and force developers to use merge requests.

By default, protected branches are designed to:

- prevent their creation, if not already created, from everybody except Maintainers
- prevent pushes from everybody except Maintainers
- prevent **anyone** from force pushing to the branch
- prevent **anyone** from deleting the branch

Read more about [protected branches](#) and [project permissions](#).

Protect a branch

Branch:

Select branch or create wildcard

Wildcards such as `*-stable` or `production/*` are supported

Allowed to merge:

Select

Allowed to push:

Select

Only groups that [have this project shared](#) can be added here

Protect

Protected branch (1)	Last commit	Allowed to merge	Allowed to push	
master <span>default</span>	0294d857 3 months ago	Developers + Mai...	No one	<div>Unprotect</div>

Par défaut dans GitLab, seuls les "maintainers" peuvent faire un push vers master.

On peut aller plus loin: ci-contre, "git push origin master" n'est plus autorisé!



# Merge requests

## Ajout de collaborateurs

GitLab

Projects

Groups

Activity

Milestones

Snippets

Search or jump to...

11

6

C code

Project

Repository

Issues 0

Merge Requests 1

CI / CD

Security & Compliance

Operations

Wiki

Snippets

Settings

General

Members

Integrations

Repository

CI / CD

Operations

Audit Events

Project members

You can invite a new member to **code** or invite another group.

Invite member

Invite group

GitLab member or Email address

Search for members to update or invite

Choose a role permission

Guest

Read more about role permissions

Access expiration date

Expiration date

Add to project

Import

Existing members and groups

Members of **code** 3

Find existing members by name

Sort by

Name, ascending

Adrien Crovato @acrovato

Given access just now

Developer

Expiration date

Boman Romain @R.Boman

Given access 3 months ago

It's you

Maintainer

Papeleux Luc @L.Papeleux

Given access 1 minute ago

Developer

Expiration date

Roles:

- guest: ne voit pas le code (très peu de droits).
- reporter: droit de lecture et de fork.
- **developer: droit lecture/écriture.**
- maintenir: droit admin (configuration du repo.)
- owner: droit suppression/chgt de nom du repo.





# Merge requests

## Configuration des Merge Requests

code

Project

Repository

Issues

Merge Requests

CI / CD

Security & Compliance

Operations

Wiki

Snippets

Settings

General

Members

Integrations

Repository

CI / CD

Operations

Audit Events

Merge requests

Choose your merge method, merge options, merge checks, and set up a default description template for merge requests.

Merge method

This will dictate the commit history when you merge a merge request

0

☐ Merge commit

Every merge creates a merge commit

1

☒ Merge commit with semi-linear history

Every merge creates a merge commit  
Fast-forward merges only  
When conflicts arise the user is given the option to rebase

☐ Fast-forward merge

No merge commits are created  
Fast-forward merges only  
When conflicts arise the user is given the option to rebase

Merge options

Additional merge request capabilities that influence how and when merges will be performed

☐ Merge pipelines will try to validate the post-merge result prior to merging  
Pipelines need to be configured to enable this feature.

☐ Automatically resolve merge request diff discussions when they become outdated

☒ Show link to create/view merge request when pushing from the command line

Merge checks

These checks must pass before merge requests can be merged

☐ Pipelines must succeed  
Pipelines need to be configured to enable this feature.

☐ All discussions must be resolved

Default description template for merge requests

L'option par défaut ne garantit pas un "fast-forward merge". Autrement dit, la version de la branche que l'on veut merger avec master peut être "en retard" par rapport à master!  
⇒ C'est dangereux!

Les 2 autres options définissent si on veut ou non un commit de merge (qui sera vide mais qui peut faciliter la lecture de l'arbre des révisions).





# Merge requests

## Ajout de "reviewers" de Merge Requests sur GitLab Enterprise (\$)

C

code

Project

Repository

Issues0

Merge Requests1

CI / CD

Security & Compliance

Operations

Wiki

Snippets

Settings

General

Members

Merge request approvals

Collapse

Set a number of approvals required, the approvers and other approval settings. [Learn more about approvals.](#)

Add approvers

Name	Members	No. approvals required	
approvers		1	<div>Edit</div> <div></div>

Add approvers

☐ Require approval from code owners ?

☐ Can override approvers and approvals required per merge request ?

☒ Remove all approvals in a merge request when new commits are pushed to its source branch

☒ Prevent approval of merge requests by merge request author ?

☐ Prevent approval of merge requests by merge request committers ?

Save changes

Définir qui va accepter les merge requests et le nombre de reviews nécessaires (ici 1 seule).

D'autres options peuvent être activées si on est dans un environnement +/- "hostile".

Décocher cette case; sinon le créateur du merge request peut définir 0 reviewer et merger sa branche directement.



# Merge requests

A ce stade, le repository est configuré pour **protéger master\* vis-à-vis de tout le monde**.

Attention, le système proposé ici requiert tout de même une **confiance importante** entre développeurs.

L'idée est:

- d'éviter des erreurs ("oops, j'ai mergé ma branche dans master sans le faire exprès..."),
- de temporiser les modifications ("je pars en vacances demain, je vais vite merger ma branche ni vu ni connu..."),
- de permettre une discussion sur les modifications du code.

**Si le degré de confiance n'est pas suffisant (externe, étudiant TFiste, débutant, interface chaise/écran/clavier déficiente)**, il est possible d'ajouter le collaborateur en tant que "Reporter" et le forcer à faire un "fork". Il pourra ainsi travailler sur une copie du repository et effectuer un Merge Request mais il ne pourra pas merger lui-même.

*\*Remarque: Le système de MR peut-être également utilisé pour documenter, reviewer, discuter n'importe quel "git merge". On se concentre ici sur la modification de master qui est la version de référence du code.*



# Merge requests

## Création d'un merge request vers master (1/4)

The screenshot shows the GitLab web interface for a project named 'Boman Romain'. The left sidebar contains navigation links: Project, Repository, Issues (0), Merge Requests (0), CI / CD, Operations, Wiki, Snippets, and Settings. The 'Merge Requests' link is circled in red. A large red arrow points from this link to a green button labeled 'New merge request' at the bottom right of the main content area, which is also circled in red. The main content area has a header 'Boman Romain > code > Merge Requests' and a notification 'You pushed to boman 1 minute ago'. Below this is an illustration of a computer screen with code and a speech bubble. Text below the illustration explains the purpose of merge requests: 'Merge requests are a place to propose changes you've made to a project and discuss those changes with others. Interested parties can even contribute by pushing commits if they want to.' A 'Create merge request' button is visible in the top right of the main content area.



# Merge requests

## Création d'un merge request vers master (2/4)

GitLab

Projects ▾ Groups ▾ Activity Milestones Snippets

+ ▾ This project Search 🔍

C

Home

Documents

Repositories

Issues

Wiki

Settings

Boman Romain > code > Merge Requests

New Merge Request

Source branch

R.Boman/code **boman**

merge boman into master  
Romain Boman authored 49 minutes ago  
28e5f68d

Target branch

R.Boman/code **master**

Merge branch 'master' of  
gitlab.uliege.be:R.Boman/code  
Romain Boman authored 3 days ago  
0294d857

Compare branches and continue

indiquer la branche  
source et la branche  
cible



# Merge requests

## Création d'un merge request vers master (3/4)

**New Merge Request**  
From `boman` into `master`

Title: Merge request Demo

*Start the title with **WIP:** to prevent a **Work In Progress** merge request from being merged before it's ready.*

Add [description templates](#) to help your contributors communicate effectively!

Description

**Write** Preview

# Merge request demo

This merge request .

[Markdown and quick actions](#)

Préfixez le titre par "WIP:" si vous préparez votre merge request (c'est un travail long!).

Réfléchir à un titre approprié!

Décrire ici en détail toutes les modifications faites.

- Nouvelles fonctionnalités? Quelle implémentation?
- Expliquez avec des figures/captures d'écran.
- Des bugs ont-ils été corrigés?
- Y a-t-il de nouveaux tests? Où en est la doc?
- Des idées pour la suite?

Lâchez-vous: plus c'est détaillé mieux c'est!



# Merge requests

## Création d'un merge request vers master (4/4)

GitLab

Projects

Groups

Activity

Milestones

Snippets

Search or jump to...

11

6

Ccode

Project

Repository

Issues0

Merge Requests0

CI / CD

Security & Compliance

Operations

Wiki

Snippets



Settings

Cross-project dependencies

Enter merge request URLs or references (e.g. path/to/project!merge\_request\_id)

List the merge requests that must be merged before this one.

Approvers

Name	Members	No. approvals required
approvers	 	1

Source branch

boman

Target branch

master

Change branches

☐ Delete source branch when merge request is accepted.

☐ Squash commits when merge request is accepted.

Submit merge request

Commits 6Changes 4

Showing 4 changed files with 5 additions and 0 deletions

.gitignore

...	@@ -3,3 +3,5 @@	...	@@ -3,3 +3,5 @@
3	*.tmp	3	*.tmp
4	*.zip	4	*.zip
5	*.bak	5	*.bak
		6	+ *.json
		7	+ *.vscode

Regarder le détail des modifs du code avant d'appuyer sur SUBMIT!  
=> Si nécessaire, faire un commit de nettoyage.



# Merge requests

Après avoir appuyé sur "submit":



# Merge requests

Vue "reviewer":

Le reviewer peut approuver le merge request avec ce bouton.

Mais il doit d'abord lire et discuter les modifications (c'est un vrai travail! Il sera co-responsable des problèmes s'il y en a plus tard).

Il peut discuter ici avec les autres développeurs (c'est un "chat" où tous les événements seront listés chronologiquement)

GitLab Projects Groups Activity Milestones Snippets Search or jump to...

Boman Romain > code > Merge Requests > 12

Open Opened 10 minutes ago by Boman Romain Edit Close merge request

### Merge request demo

Request to merge boman into master Open in Web IDE Check out branch

Approve Requires approval from approvers. View eligible approvers

Merge You can only merge once the items above are resolved

You can merge this merge request manually using the command line

Discussion 0 Commits 6 Changes 4 Show all activity

Write Write a comment or your files here...

Markdown and quick actions are supported Attach a file

Comment Close merge request

To Do Mark as done

0 Assignees None - assign yourself Edit

Milestone None Edit

Time tracking No estimate or time spent

Labels None Edit

Lock merge request Unlocked Edit

2 participants

Notifications

Reference: R.Boman/code!2





# Merge requests

Discussion 1 **Commits 6** Changes 4

0/1 thread resolved

23 Aug, 2019 3 commits

**merge boman into master**  
Boman Romain authored 6 months ago

28e5f68d

**update .gitignore**  
Boman Romain authored 6 months ago

1c9802e8

**add .vscode to ignore list**  
Boman Romain authored 6 months ago

e477a69a

19 Aug, 2019 2 commits

**add doc for solver.py**  
Boman Romain authored 6 months ago

3717391f

**add doc to fct**  
Boman Romain authored 6 months ago

83f0f437

18 Aug, 2019 1 commit

**improved README**  
Boman Romain authored 6 months ago

8abbbe5f



# Merge requests

Discussion 1

Commits 5

**Changes 4**

0/1 thread resolved

↔

Changes between

latest version ▾

and

master ▾

4 Files

+5 -0

Expand all

⚙ ▾

▼ .gitignore

+2 -0

[↑ Show all lines](#)

3

\*.tmp

4

\*.zip

5

\*.bak

6

+ \*.json

7

+ .vscode

> README.md

+1 -0

This diff is collapsed. [Click to expand it.](#)

> solver.py

+1 -0

This diff is collapsed. [Click to expand it.](#)

> tools/fct.py

+1 -0

This diff is collapsed. [Click to expand it.](#)



# Merge requests

Vue "reviewer":

GitLab Merge Requests interface showing a merge request for the `code` project. The interface displays the diff view for the `latest version` and `master` branches, showing changes to files like `tools/fct.py`, `.gitignore`, `README.md`, and `solver.py`. A context menu is open over a line in the `README.md` diff, showing options to **Start a review** or **Add comment now**. The right sidebar shows metadata including assignees, milestones, time tracking, labels, lock status, participants, and notifications.

Lors de sa relecture, le reviewer peut cliquer sur n'importe quelle ligne pour ajouter un commentaire et démarrer un fil de discussion sur la portion de code concernée.



# Merge requests

La discussion continue jusqu'à résolution du problème...

Discussion 2

Commits 6

Changes 4

Show all activity ▾

1/1 thread resolved

**Papeleux Luc** @L.Papeleux started a thread on an old version of the diff 2 months ago  
Resolved by Boman Romain just now

Toggle thread

M+ README.md

3 3 August 2019

**Papeleux Luc** @L.Papeleux · 2 months ago

Are you sure about the date (Aug 2019)???

Developer ✓ 😊 ✎ ⋮

**Boman Romain** @R.Boman · just now

Thanks! I am going to fix it!

Maintainer ✓ 😊 ✎ ⋮

**Boman Romain** @R.Boman changed this line in [version 2 of the diff](#) just now

Reply...

Unresolve thread

**Boman Romain** @R.Boman added 1 commit just now

- [6e51f7cb](#) - fix date

[Compare with previous version](#)

**Boman Romain** @R.Boman resolved all threads just now

L'auteur du merge request peut corriger les problèmes en ajoutant des commits à la branche du merge request



# Merge requests

Une fois que tout est réglé, le reviewer appuie sur "Approve"

Open

Opened 2 months ago by Boman Romain

Edit

Close merge request

## Merge request demo

Request to merge **boman** into **master**

Open in Web IDE

Check out branch

Merge request approved. Approved by

> [View eligible approvers](#)

Merge

☐ Delete source branch

☐ Squash commits

> **7 commits** and **1 merge commit** will be added to master. [Modify merge commit](#)

You can merge this merge request manually using the [command line](#)

Il ne reste plus qu'à appuyer sur "merge" et, généralement, supprimer la branche source.





# Merge requests

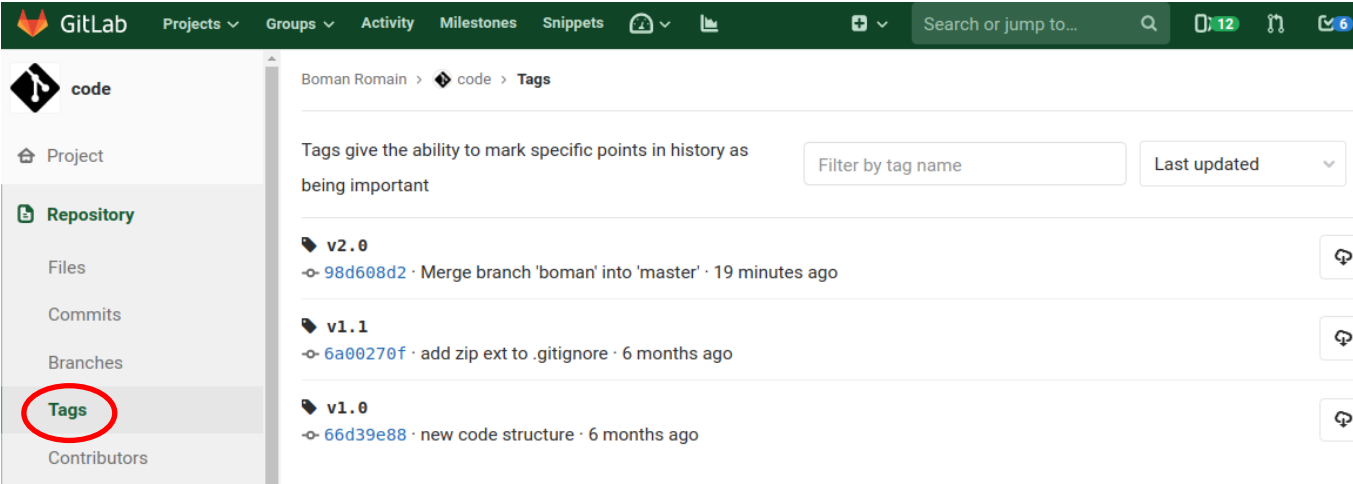
## Création d'un tag / d'une Release

Pour conserver une référence claire vers le merge request, il peut être intéressant de créer un tag:

```
git tag v2.0
git push origin --tags
```

GitLab permet également de créer une "Release" à partir d'un tag. Il s'agit d'un tag enrichi (par du texte, des fichiers, etc).

Cliquez ici pour enrichir le tag. Il sera mis en avant avec une description, des assets, etc.



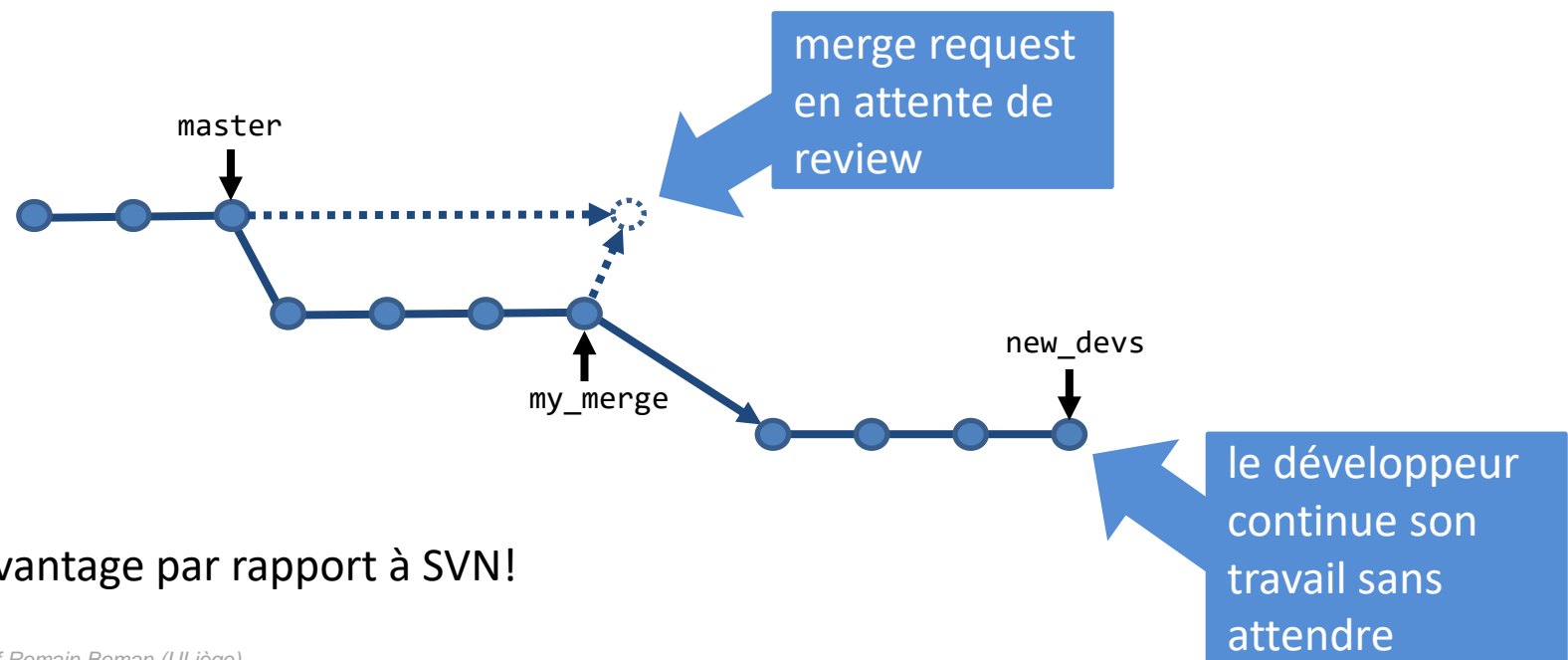


# Merge requests

**Remarque: "cette procédure va me ralentir!" (1/2)**

Ce processus de review peut être long. Ce n'est **pas un problème!**

Le développeur qui voudrait continuer ses développements en attendant l'acceptation de son merge request peut créer une **deuxième branche** à partir de son merge request et continuer son travail, "comme si" son merge request était accepté.



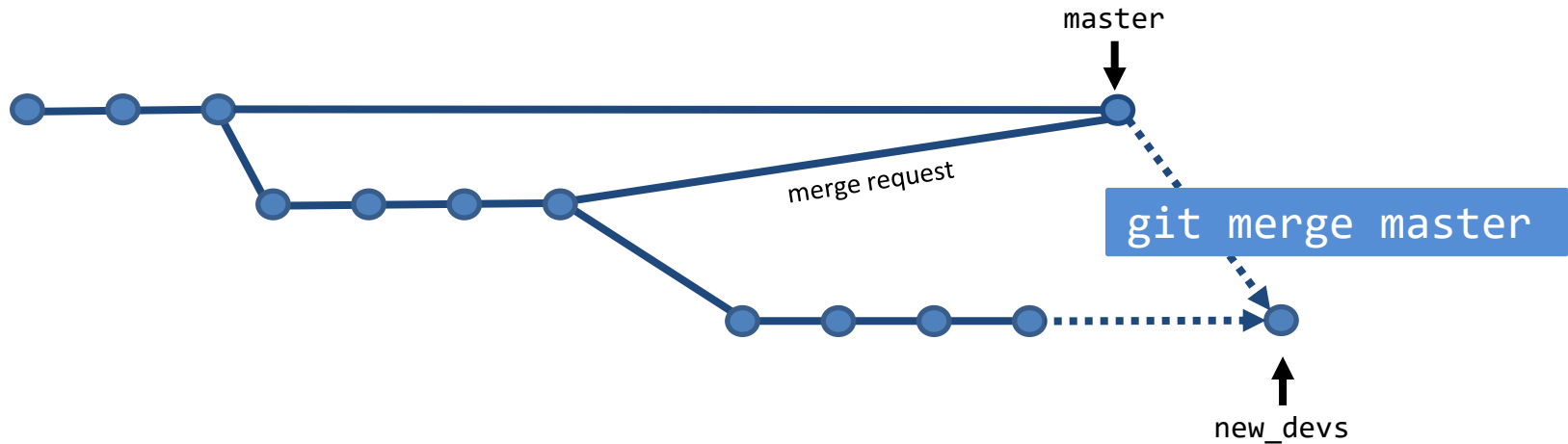
Gros avantage par rapport à SVN!



# Merge requests

**Remarque: "cette procédure va me ralentir!" (2/2)**

Lorsque le MR est accepté:



Conséquence:

L'argument "accepte mon merge request, parce que je suis bloqué", n'est pas valable.





# Merge requests

## Problème: nettoyage avant un merge request

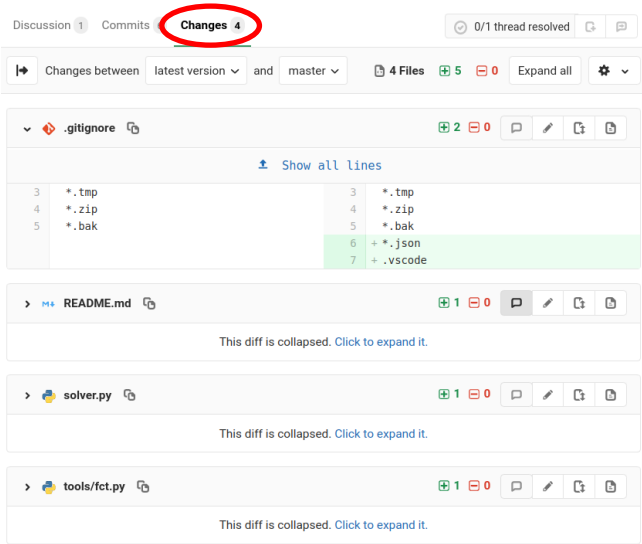
Ma branche boman est en avance sur origin/master et je suis prêt à faire un merge request. J'aimerais produire un merge request de qualité, mais j'ai commité du code pas très propre tout au long de mes développements. J'aimerais le **nettoyer**.

De plus, je ne suis pas sûr d'avoir en tête tout ce que j'ai fait. J'ai tardé à faire un merge request et j'ai du mal à **documenter** toutes mes modifs pour la description du merge request.

Autrement dit, j'aimerais repasser sur toutes mes modifs par rapport à master avant de créer un merge request.

Comment obtenir cette vue là dans Visual Studio Code avant de faire un merge request?

Utile également pour un reviewer si GitLab devient lent (car trop de modifs!)

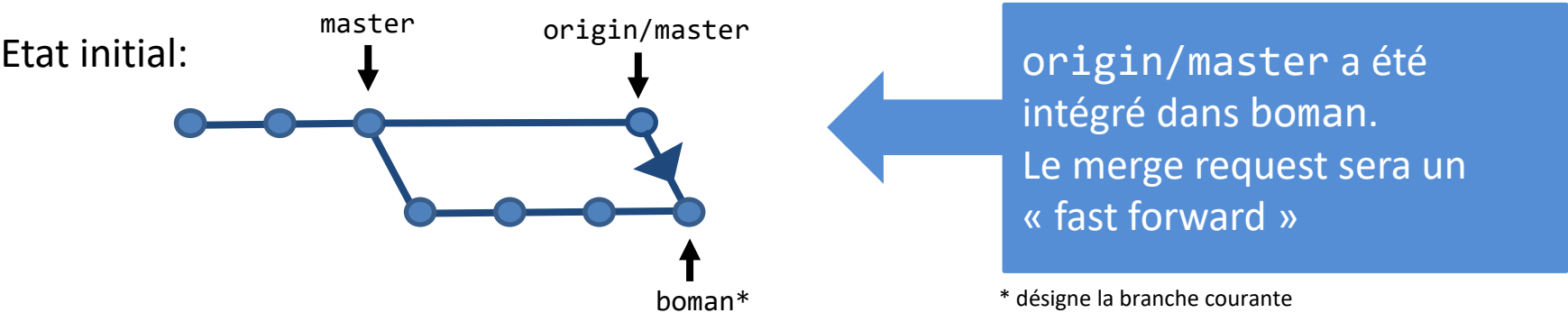




# Merge requests

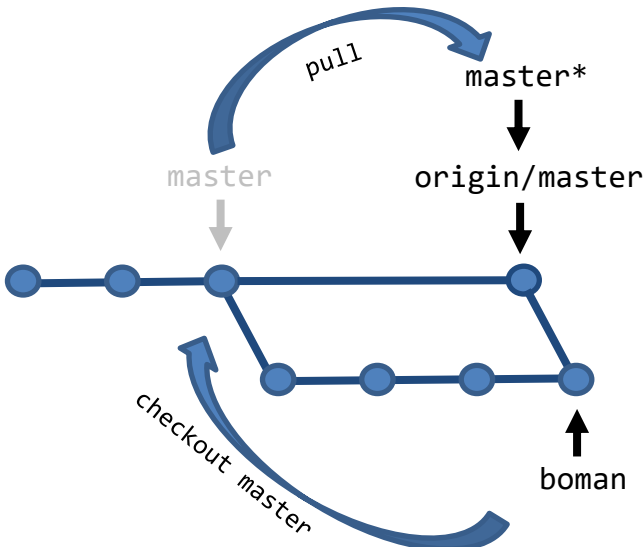
## Solution: nettoyage avant un merge request (1/6)

Un moyen est de créer une "branche de nettoyage".



Etape 1: On met à jour master qui n'a pas évolué depuis le début du travail.

```
git checkout master
git pull
```



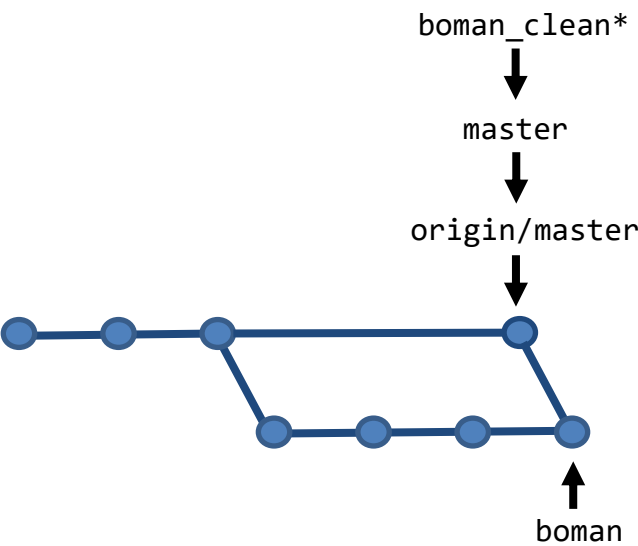


# Merge requests

***Solution: nettoyage avant un merge request (2/6)***

Etape 2: On crée un branche (locale) à partir de là

```
git checkout -b boman_clean
```

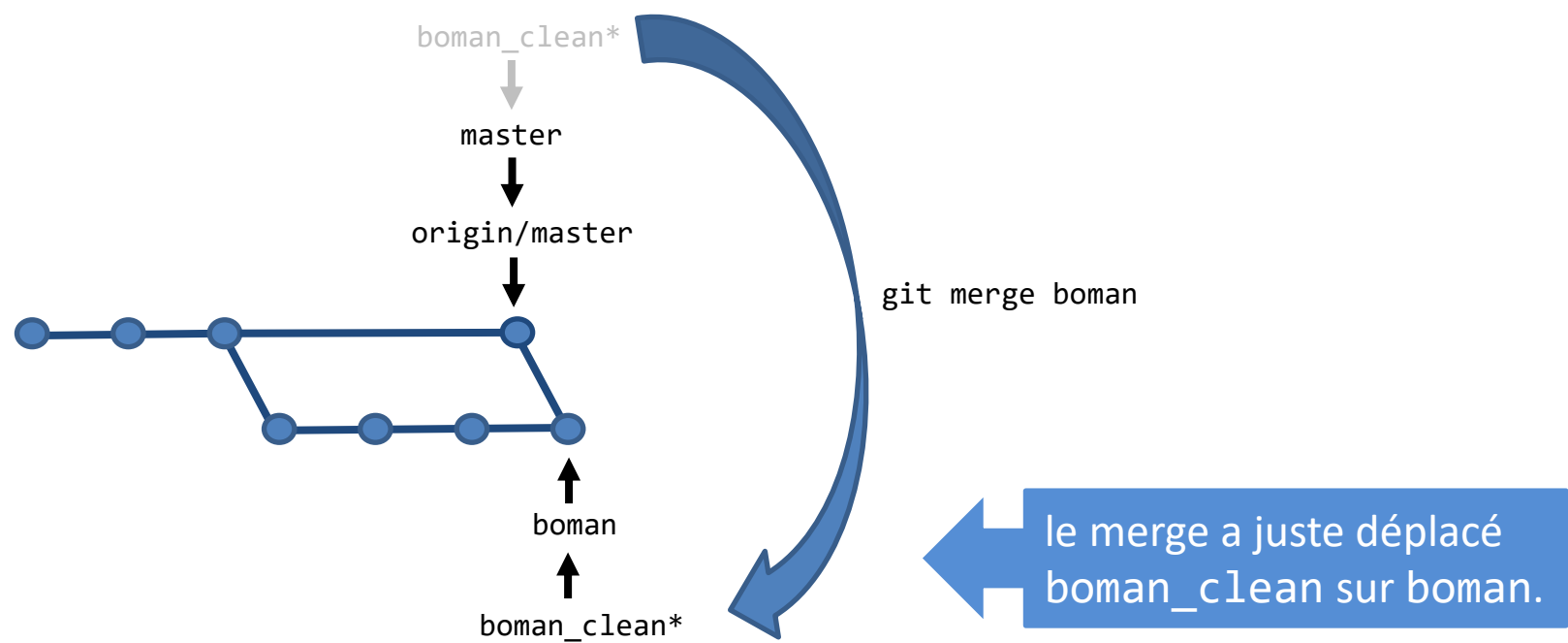




# Merge requests

## *Solution: nettoyage avant un merge request (3/6)*

On aimerait merger la branche boman dans boman\_clean pour voir les différences entre ces 2 branches.  
Puisqu'il s'agit d'un "fast-forward merge", un simple "git merge boman" ne va pas nous aider.



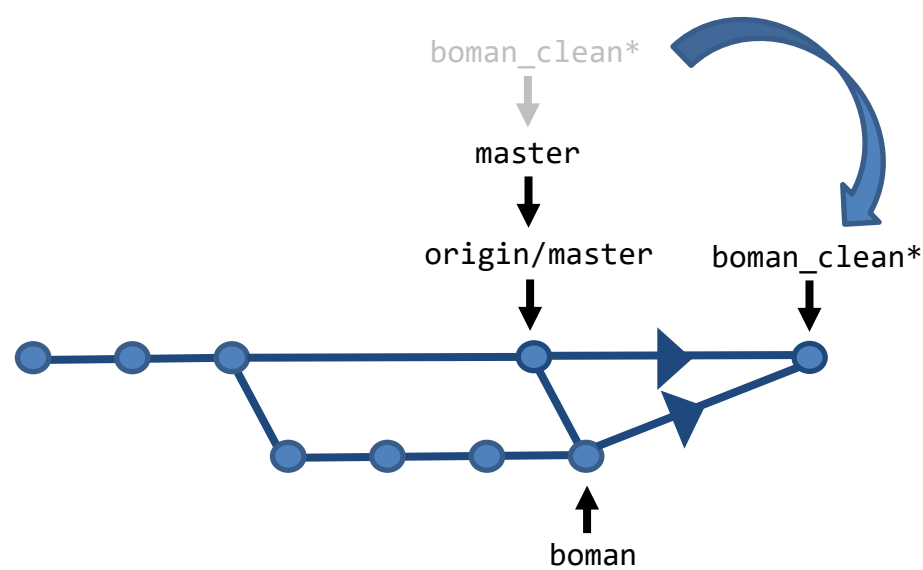


# Merge requests

## *Solution: nettoyage avant un merge request (4/6)*

Etape 3: Il faut dire à git de ne pas faire de fast forward (--no-ff) et de ne pas faire de commit (--no-commit)

```
git merge boman --no-ff --no-commit
```



Les fichiers sur disque se retrouvent dans un état où toutes les modifs de tous les commits de la branche boman sont appliqués à master d'un seul coup.  
On peut ainsi analyser les modifications avec `git diff` (ou VS Code).



# Merge requests

## *Solution: nettoyage avant un merge request (5/6)*

Pour nettoyer, il suffit de parcourir les différences et faire un commit...

The screenshot shows the Visual Studio Code interface. The left sidebar displays the 'STAGED CHANGES' section with 4 files: `.gitignore`, `README.md`, `solver.py`, and `fct.py tools`. The main editor window shows the `.gitignore` file with the following content:

```
1 *.pyc
2 *~
3 *.tmp
4 *.zip
5 *.bak
6
7+ *.json
7+ .vscode
8
```

The terminal at the bottom shows the command `git merge --no-ff --no-commit boman` and the output `Automatic merge went well; stopped before committing as requested`.



# Merge requests

## ***Solution: nettoyage avant un merge request (6/6)***

Cette vue permet de nettoyer et écrire la description du merge request.

Si modif: nettoyer, commiter et push vers origin

```
[nettoyages]
git add .
git commit -m "cleaning"
git checkout boman
git merge boman_clean
git push
```

Si pas de modif: supprimer la branche locale:

```
git merge --abort
git checkout boman
git branch -d boman_clean
```



# Issues

## *Lister les problèmes, les bugs, les feature requests*

GitLab permet de définir des "**issues**". Il s'agit de  **fils de discussion**  relatifs à un bug, une nouvelle fonctionnalité. Le fonctionnement est similaire à un forum.

Les issues sont **numérotées** (comme les merge requests) et peuvent être facilement référencées ailleurs dans des discussions GitLab (avec le signe #: par exemple #4 est l'issue numéro 4).

Elles possèdent aussi des **labels** tels "bugs", "question", "enhancement", etc. en fonction de leur type.

L'issue est donc un lieu de **discussion** privilégié pour parler de l'évolution du code, de ses problèmes, etc.

On peut **interpeler** n'importe quel développeur en mentionnant son nom (avec le signe @, comme sur facebook)

Les issues peuvent être assignées à un développeur et jouer le rôle de **TODO list**.





# Issues

## Exemple de liste d'issues d'un projet plus actif que le nôtre

Projects ▾ Groups ▾ Activity Milestones Snippets

Search or jump to...

12

6

?

waves

Project

Repository

**Issues** 11

List

Boards

Labels

Milestones

Merge Requests 0

CI / CD

Security & Compliance

Operations

Packages

Wiki

Snippets

Aerospace and Mechanical Engineering > waves > Issues

Open 11 Closed 17 All 28

Search or filter results...

Created date

Edit issues

New issue

**Deprecation warnings for Intel tbb 2020**  
#46 · opened 1 month ago by Adrien Crovato

0

updated 1 month ago

**Aliasing warnings for windows**  
#45 · opened 1 month ago by Adrien Crovato cleaning

0

updated 1 month ago

**use MKL if MKL and BLAS are present**  
#44 · opened 1 month ago by Boman Romain cleaning enhancement

2

updated 1 month ago

**Mesh deformation**  
#43 · opened 1 month ago by Adrien Crovato bug

0

updated 1 month ago

**Find a name for flow**  
#42 · opened 6 months ago by Boman Romain question

4

updated 1 month ago

**Change the name of mrstlnos**  
#37 · opened 10 months ago by Boman Romain

3

updated 10 months ago

**Trilinos compilation warnings**  
#28 · opened 1 year ago by Boman Romain cleaning

0

updated 1 year ago

**Mem**

0



# Tester le code: Pipelines

## *Intégration continue (GitLab CI)*

GitLab permet d'exécuter des scripts lors de différents événements liés au repository (push, merge request, tag, etc.)

Il s'agit généralement de tester la compilation du code, le lancement de batterie de tests de non régression, la génération de documentation, la création d'une version binaire installable, la mise à jour du code sur un serveur "de production", etc.

On parle d'**intégration continue**.

Ces scripts sont exécutés sur des "**runners**". Ce sont des machines sur le réseau préconfigurées de manière bien définie pour permettre l'exécution des scripts. En pratique, ces configurations sont des images "docker".

Les scripts sont définis dans un fichier nommé `.gitlab-ci.yml` à la racine du projet.

Les scripts peuvent être agencés sous forme d'un **pipeline**, c'est-à-dire une série de tâches qui s'exécutent en série ou en parallèle suivant leurs dépendances respectives.



# Tester le code: Pipelines

## Exemple de fichier `.gitlab-ci.yml`

"test" est le nom de l'étape de notre pipeline (qui n'en comporte qu'une)

On utilise l'image docker `python:2`, c'est à dire un container qui contient tout ce qu'il faut pour lancer python:  
[https://hub.docker.com/\\_/python](https://hub.docker.com/_/python)

```
test:
  image: python:2
  script:
    - python solver.py 0.1
```

Notre script consiste à lancer `solver.py` avec l'argument `0.1`.  
Si le script ne plante pas, l'étape "test" sera considérée comme réussie.



# Tester le code: Pipelines

## Dans GitLab (1/2)

```
git add .gitlab-ci.yml
git commit -m "add continuous integration"
git push
```

GitLab

Projects ▾ Groups ▾ Activity Milestones Snippets

Search or jump to...

12 6 ?

code

Project

Repository

Issues 0

Merge Requests 0

**CI / CD**

**Pipelines**

Jobs

Schedules

Charts

Boman Romain > code > Pipelines

All 1 Pending 0 Running 0 Finished 1 Branches Tags

Run Pipeline Clear Runner Caches CI Lint

Status	Pipeline	Triggerer	Commit	Stages
passed	#738 latest		master → 9e6db793 add gitlab CI	00:00:19 just now

Le pipeline vient d'être exécuté et il est passé avec succès.

Un clic sur ce bouton permet de voir la console du runner qui l'a exécuté...



# Tester le code: Pipelines

## Dans GitLab (2/2)

The screenshot displays the GitLab CI/CD interface for a project named 'code'. The top navigation bar includes 'Projects', 'Groups', and 'More' menus, along with a search bar and user profile. The main content area shows the details of 'Job #2796', which was triggered 2 hours ago by 'Boman Romain'. The job status is 'passed'. The terminal output shows the job running on 'Runner 2 (741)' using a Docker executor with a python:2 image. The output includes cloning the repository, checking out commit '9e6db793', and running a script 'python solver.py 0.1' which outputs 'sin(0.100000) = 0.099833'. The job concludes with 'Job succeeded'. On the right, the 'test' stage is detailed, showing a duration of 19 seconds, a timeout of 1h, and the runner 'Runner 2 (741) (#6)'. A blue arrow points from the 'Runner 2' text in the stage details to a callout box. Another blue arrow points from the 'Job succeeded' status to another callout box.

Notre code a été exécuté avec succès!

Il a été exécuté sur "Runner 2", un runner offert par le SeGI. Il est possible d'ajouter ses propres runners.



# Tester le code: Pipelines

## Que se passe-t-il quand on introduit un bug? (1/3)

Après avoir introduit un bug, le pipeline ne passe plus. l'utilisateur reçoit un mail:

Boîte de réception

★ Messages suivis

🕒 En attente

➡ Important

💬 Tous les chats

➤ Messages envoyés

📄 Brouillons

✉ Tous les messages

🚫 Spam 2

🗑 Corbeille

📁 Catégories

👥 Réseaux sociaux

📢 Notifications 30

📖 Forums

📄 Promotions 1

🤖 Bots 38

code | Pipeline #743 has failed for master | 9603c29d

Boîte de réception x

GitLab

À

16:29 (il y a 1 minute)

☆ ↶ ⋮

✖ Your pipeline has failed.

Project	Boman Romain / code
Branch	🔗 master
Commit	🔗 9603c29d test pipeline failure
Commit Author	Boman Romain



# Tester le code: Pipelines

## Que se passe-t-il quand on introduit un bug? (2/3)

Le commit est marqué par une croix rouge (au lieu d'un "V" vert):

Projects ▾ Groups ▾ More ▾

▾ 12 6 ▾ ▾

≡

Boman Romain > code > Commits

master ▾ code

12 Mar, 2020 2 commits

**test pipeline failure**  
Boman Romain authored 4 minutes ago

9603c29d

**add gitlab CI**  
Boman Romain authored 4 hours ago

9e6db793

11 Mar, 2020 2 commits

**Merge branch 'boman' into 'master'**

98d608d2

**fix date**  
Boman Romain authored 1 day ago

6e51f7cb



# Tester le code: Pipelines

*Que se passe-t-il quand on introduit un bug? (3/3)*

**test** Retry

New issue

Duration: 38 seconds  
Timeout: 1h (from project) ?  
Runner: Runner 3 (742) (#7)

Commit 9603c29d 🔗  
test pipeline failure

✖ Pipeline #743 for master

test

→ ✖ test

Le script a planté (et donc il a retourné un code d'erreur)

Les détails de l'erreur sont visibles dans la console.





# Tester le code: Pipelines

## Vérification du pipeline lors de merge requests

code

Project

Repository

Issues 0

**Merge Requests 1**

CI / CD

Security & Compliance

Operations

Wiki

Snippets

Settings

Boman Romain > code > Merge Requests > 13

Open

Opened just now by Boman Romain

### fix pipeline

The MR makes our great code work again!

Request to merge **boman** into **master**

**Pipeline #745 passed for ad9d3e7b on boman**

Requires approval from approvers.

>

View eligible approvers

Merge

You can only merge once the items above are resolved

You can merge this merge request manually using the [command line](#)

Lors d'un merge request, le pipeline est lancé et son succès est nécessaire pour pouvoir merger la branche dans master!



# Slack

## Comment interagir encore plus avec GitLab? (1/2)

Il est possible de demander à GitLab d'envoyer des notifications supplémentaires à des services web et type "chat" (il y en a des dizaines!)

⚙️ Settings

General

Members

**Integrations**

Repository

CI / CD

Operations

Audit Events

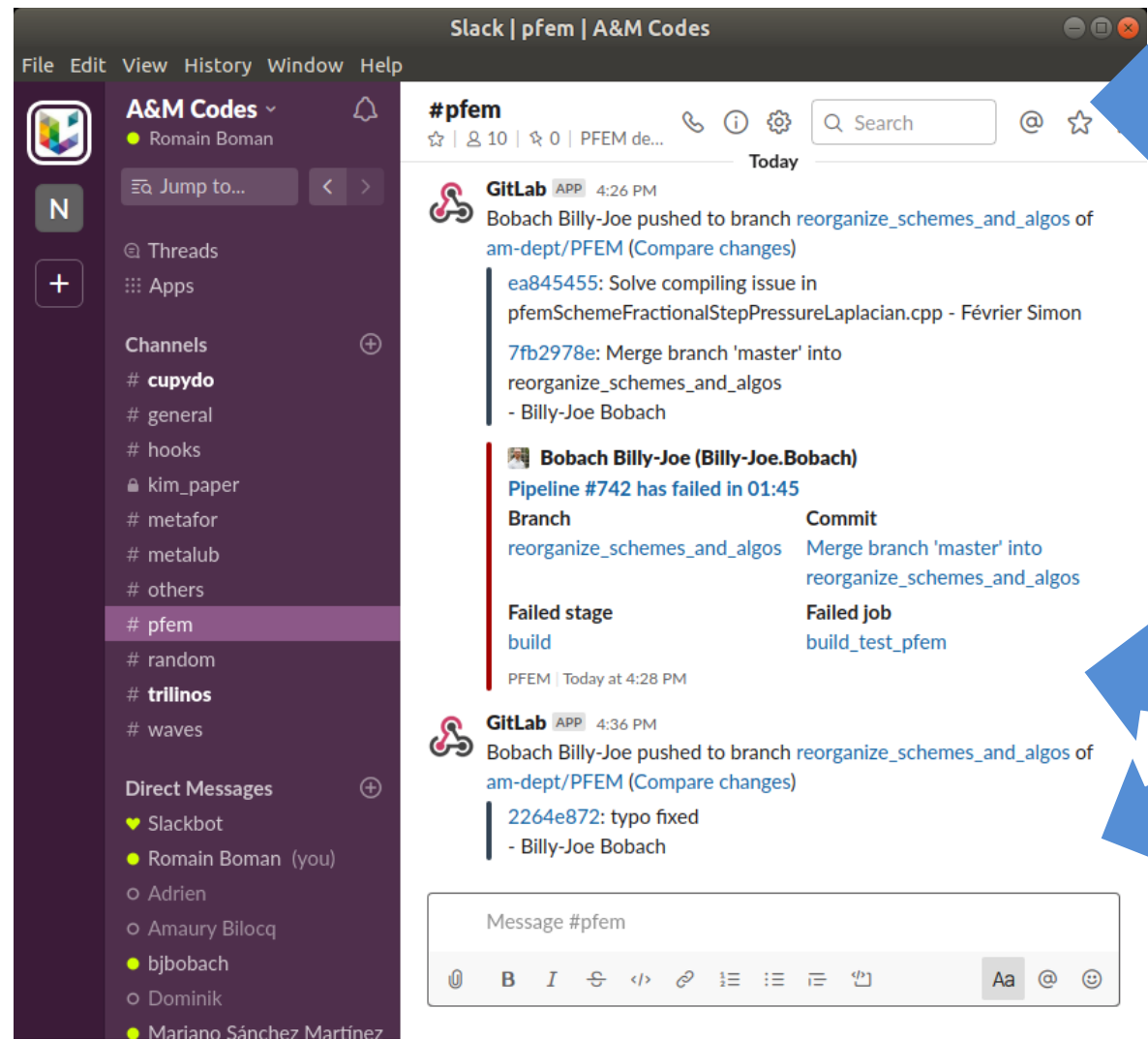
⏻ Pipelines emails	Email the pipelines status to a list of recipients.
⏻ PivotalTracker	Project Management Software (Source Commits Endpoint)
⏻ Prometheus	Time-series monitoring service
⏻ Pushover	Pushover makes it easy to get real-time notifications on your Android device, iPhone, iPad, and Desktop.
⏻ Redmine	Redmine issue tracker
⏻ <b>Slack notifications</b>	Receive event notifications in Slack
⏻ Slack slash commands	Perform common operations in Slack
⏻ YouTrack	YouTrack issue tracker

Je présente ici Slack  
<https://slack.com/>



# Slack

## Comment interagir encore plus avec GitLab? (2/2)



Slack est un système de chat ressemblant à IRC. Il permet de créer des "channels", c'est-à-dire des salons de discussion sur un thème donné (ci-contre, sur un code donné). Il permet également d'envoyer des messages privés aux autres développeurs.

GitLab peut poster des notifications dans ces salons pour la plupart des événements. Tout est configurable dans l'onglet "Integration" de GitLab

# Utiliser git

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int tmax)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        #pragma omp parallel for num_threads(nbt)
```

## Références

```
    }

    double tstart = omp_get_wtime();
    test.execute(nbt);
    double tstop = omp_get_wtime();
    double cpu = tstop-tstart;

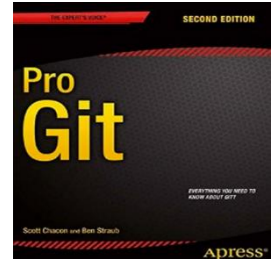
    OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

    std::cout << res;
```

# Références

## Gratuit

- <https://git-scm.com/doc> : documentation officielle
- <https://git-scm.com/book/en/v2> : Pro Git book
- <https://www.atlassian.com/fr/git> : Tutoriels Atlassian (doc bitbucket)

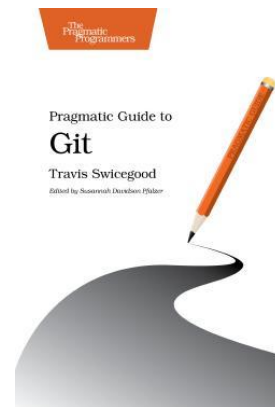


## Payant

- T. Swicegood - Pragmatic Guide to Git
- T. Swicegood - Pragmatic Version Control Using Git

## Youtube

[https://www.youtube.com/results?search\\_query=git](https://www.youtube.com/results?search_query=git)



# Utiliser git

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int tmax)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        #pragma omp parallel for num_threads(nbt)
```

## FAQ

(questions dans le désordre)

```
}

double tstop = omp_get_wtime();
double cpu = tstop-tstart;

OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

std::cout << res;
```



# FAQ

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int tmax)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
        for (int j=0; j<n; j++)
            c[i*n+j] = a[i*n+j] + b[i*n+j];
}
```

- 1. J'ai commité les résultats de batterie!
- 2. Supprimer les branches locales qui ont été mergées
- 3. Comment obtenir l'état de modification d'une branche par rapport à une autre?
- 4. Utiliser PUTTY au lieu de OpenSSH sous Windows

```
idx2++;
double tstart = omp_get_wtime();
test.execute(nbt);
double tstop = omp_get_wtime();
double cpu = tstop-tstart;

OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

std::cout << res;
```

# FAQ 1 - push batterie

**J'ai commité les résultats de batterie! (mais j'ai pas encore fait push)**

```
git reset HEAD^ apps/verif
```

supprime  
l'erreur

```
git commit --amend
```

fait un nouveau commit parallèle au commit "trop riche". Le contenu de ce commit est une fusion entre le dernier commit et l'état actuel.

**... et j'ai déjà fait push**

Faire la même chose, ensuite

```
git push --force
```

Si quelqu'un travaille sur la même branche, il va avoir des gros problèmes puisque l'ancienne branche va devenir "morte". Il pourrait faire alors un "push --force", lui aussi, ce qui déclarerait une guerre!



# FAQ 2 - nettoyage repo local

## Supprimer les branches locales qui ont été mergées (et supprimées du remote) (1/3)

Travailler avec un « vieux clone » peut devenir confus (branches non synchronisées ou effacées ailleurs). Comment nettoyer son répertoire de travail sans faire un nouveau git clone?

```
git remote show origin
```

```
* remote origin
Fetch URL: git@gitlab.uliege.be:am-dept/waves.git
Push URL: git@gitlab.uliege.be:am-dept/waves.git
HEAD branch: master
Remote branches:
  adrien
  adrien_amr
  boman
  feature_shrd
  master
  papeleux
  refs/remotes/origin/CMAME
  refs/remotes/origin/ci_with_reduction
  refs/remotes/origin/clean_mirrors
  refs/remotes/origin/feature_block
  refs/remotes/origin/feature_eigen
Local branches configured for 'git pull':
...
```

anciennes branches de l'index  
supprimées d'origin

Peuvent être supprimées avec  
`git remote prune origin`



```
tracked
tracked
tracked
tracked
tracked
tracked
stale (use 'git remote prune' to remove)
stale (use 'git remote prune' to remove)
stale (use 'git remote prune' to remove)
stale (use 'git remote prune' to remove)
stale (use 'git remote prune' to remove)
```

suite: voir slide suivant

# FAQ 2 - nettoyage repo local

## Supprimer les branches locales qui ont été mergées (et supprimées du remote) (2/3)

suite de... `git remote show origin`

...

Local branches configured for 'git pull':

boman	merges with remote boman
ci_with_reduction	merges with remote ci_with_reduction
feature_block	merges with remote feature_block
feature_eigen	merges with remote feature_eigen
fix_docker	merges with remote fix_docker
fix_msvc_build	merges with remote fix_msvc_build
master	merges with remote master
master_update_Trilinos	merges with remote master_update_Trilinos
master_update_Trilinos_merge_kim	merges with remote master_update_Trilinos_merge_kim
papeleux	merges with remote papeleux

toutes les branches  
locales (y compris celles  
qui pointent vers des  
branches supprimées)

Local refs configured for 'git push':

boman	pushes to boman	(up to date)
master	pushes to master	(local out of date)
papeleux	pushes to papeleux	(local out of date)

état des branches locales













# FAQ 2 - nettoyage repo local

## Supprimer les branches locales qui ont été mergées (et supprimées du remote) (3/3)

```
git branch --merged          # liste les branches locales mergées
git branch -d branch1       # supprime les branches locales 1 à 1
...
git branch -av              # affiche toutes les branches:
                             # il reste les "tracking branches"
                             # (origin/branch1 p, expl)
git remote prune origin     # supprime les tracking branches inutiles
```

# FAQ 3 - état d'une branche

## Comment obtenir l'état de modification d'une branche par rapport à une autre? (1/2)

Active branches				
 <b>adrien</b>				
 <a href="#">6cb74ada</a> · Merge remote-tracking branch 'origin/master' into adrien · 1 week a...	0	2	<a href="#">Merge request</a>	<a href="#">Compare</a>
 <b>papeleux</b>				
 <a href="#">eca34777</a> · remove msys2/mingw64 config file · 1 month ago	0	0	<a href="#">Merge request</a>	<a href="#">Compare</a>
 <b>master</b> <span>default</span> <span>protected</span>				
 <a href="#">eca34777</a> · remove msys2/mingw64 config file · 1 month ago			<a href="#">Merge request</a>	<a href="#">Compare</a>
 <b>boman</b> <span>merged</span>				
 <a href="#">00ab9dc1</a> · porting to PySide2 · 1 month ago	7	0	<a href="#">Merge request</a>	<a href="#">Compare</a>
Stale branches				
 <b>feature_shrd</b>				
 <a href="#">40171e0c</a> · Add share_ptr to tbox, heat and flow functions · 8 months ago	94	1	<a href="#">Merge request</a>	<a href="#">Compare</a>
 <b>adrien_amr</b>				
 <a href="#">3be4fb3a</a> · updated · 1 year ago	292	3	<a href="#">Merge request</a>	<a href="#">Compare</a>

La branche `adrien_amr` est 292 commits « en retard de » (behind) `origin/master` et 3 commits « en avance » (ahead)

# FAQ 3 - état d'une branche

## Comment obtenir l'état de modification d'une branche par rapport à une autre? (2/2)

- A partir de n'importe quelle branche

```
git rev-list --left-right --count origin/master...origin/adrien_amr
292      3
```

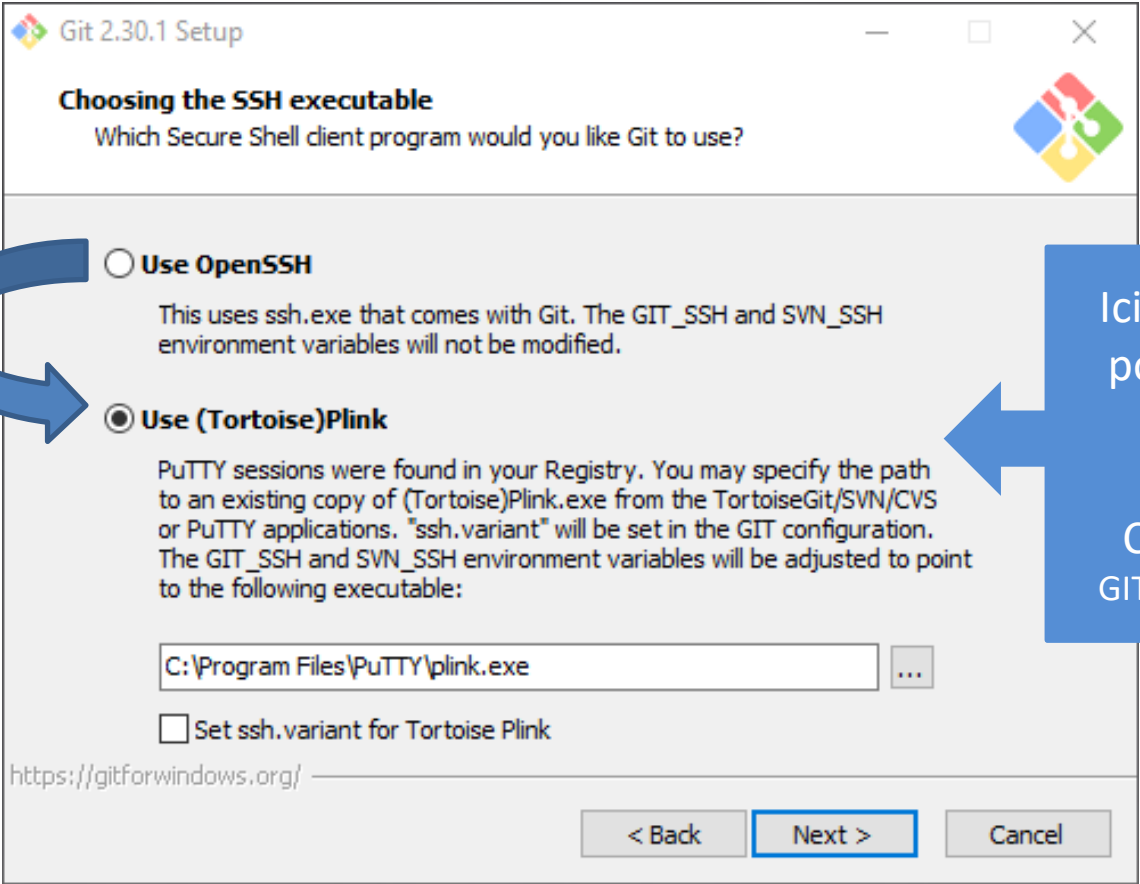
- A partir de la branche courante:

```
git checkout adrien_amr
git rev-list --left-right --count origin/master...@
292      3
```

# FAQ 4 - PUTTY

## Utiliser PUTTY au lieu de OpenSSH sous Windows

- installez d'abord PuTTY
- ensuite, lors de l'installation de git:



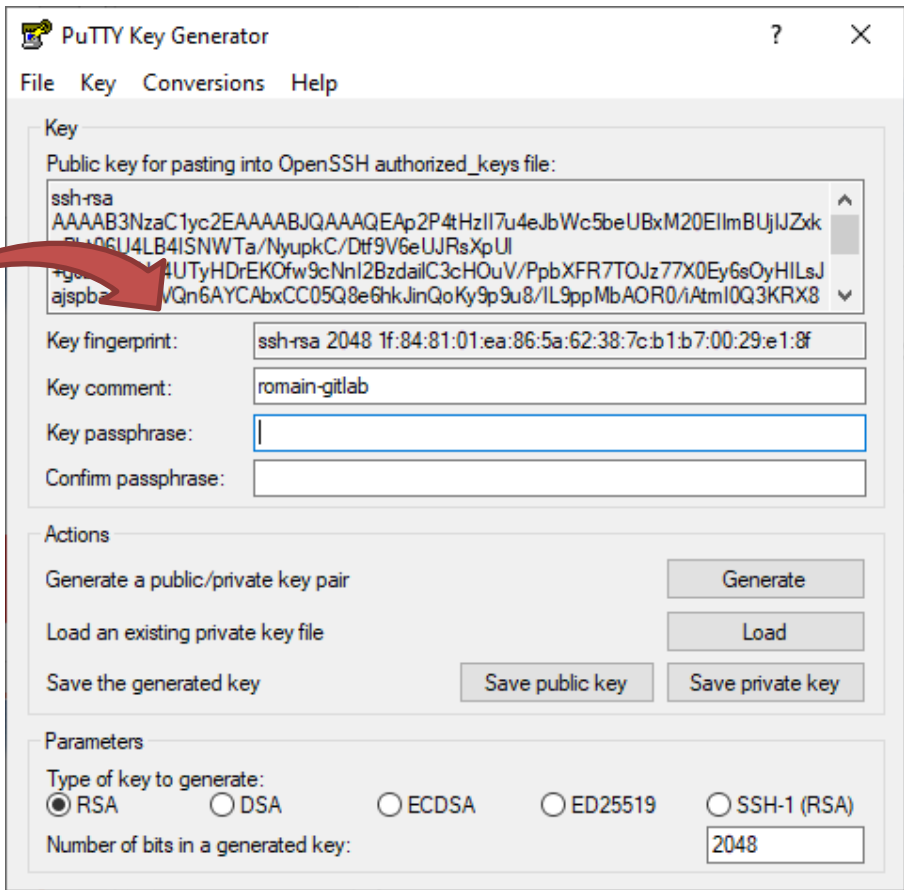
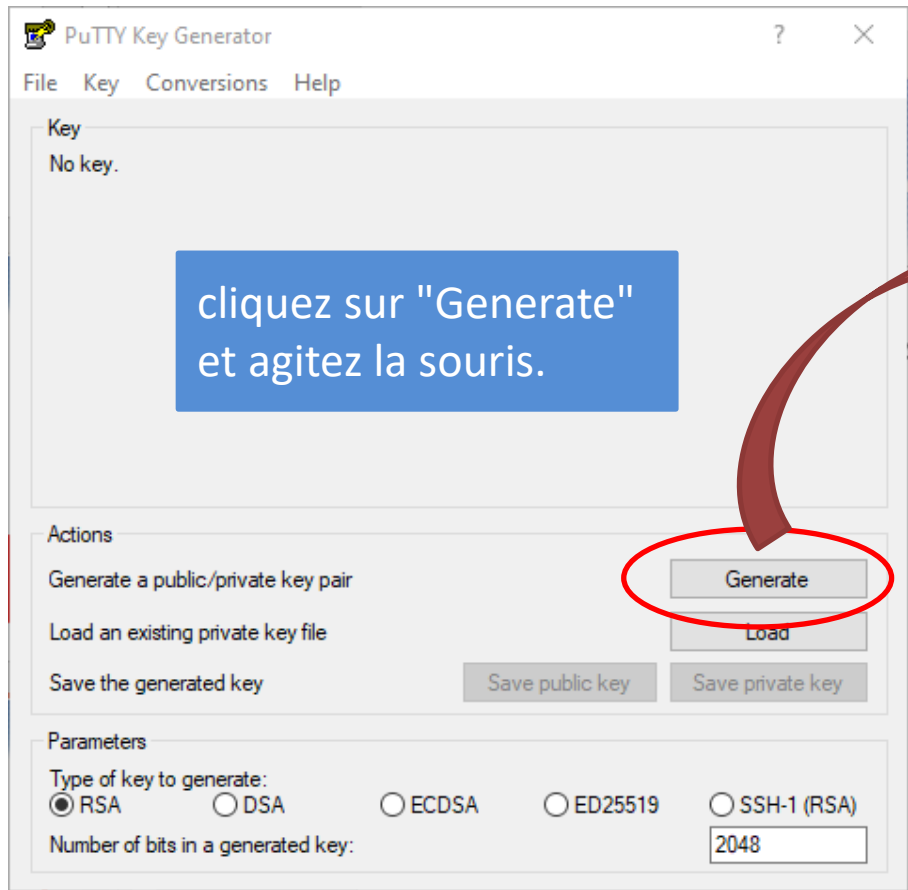
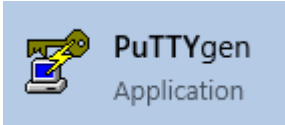
Ici, j'utilise PuTTY que j'utilise déjà pour me connecter aux machines Linux.

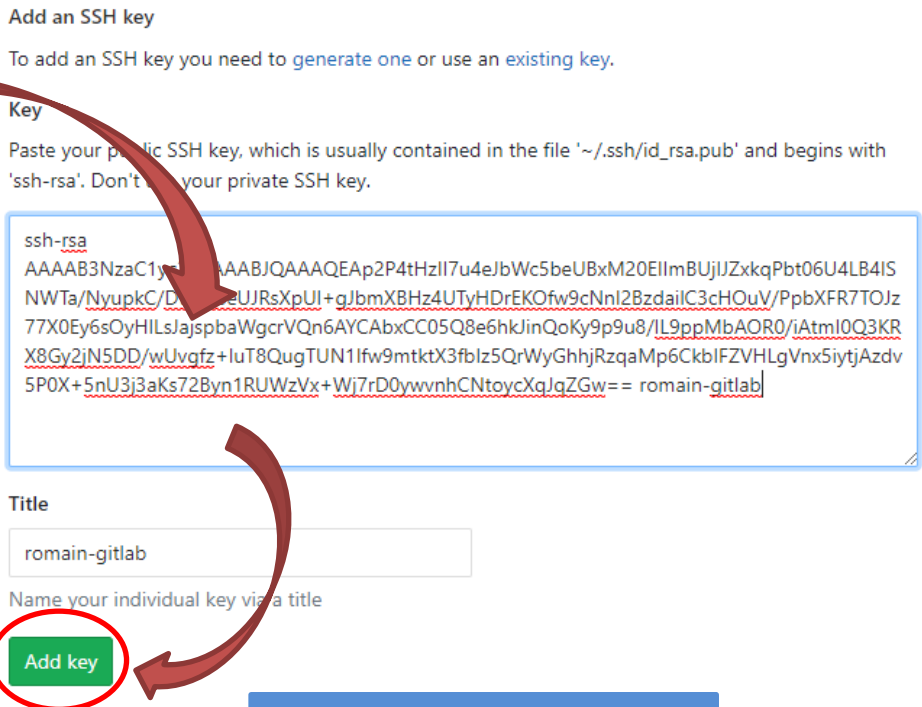
Cette option va créer la variable  
GIT\_SSH = C:\Program Files\PuTTY\plink.exe

# FAQ 4 - PUTTY

## Création d'une clef SSH sous Windows avec PuTTYgen

<https://www.putty.org/>





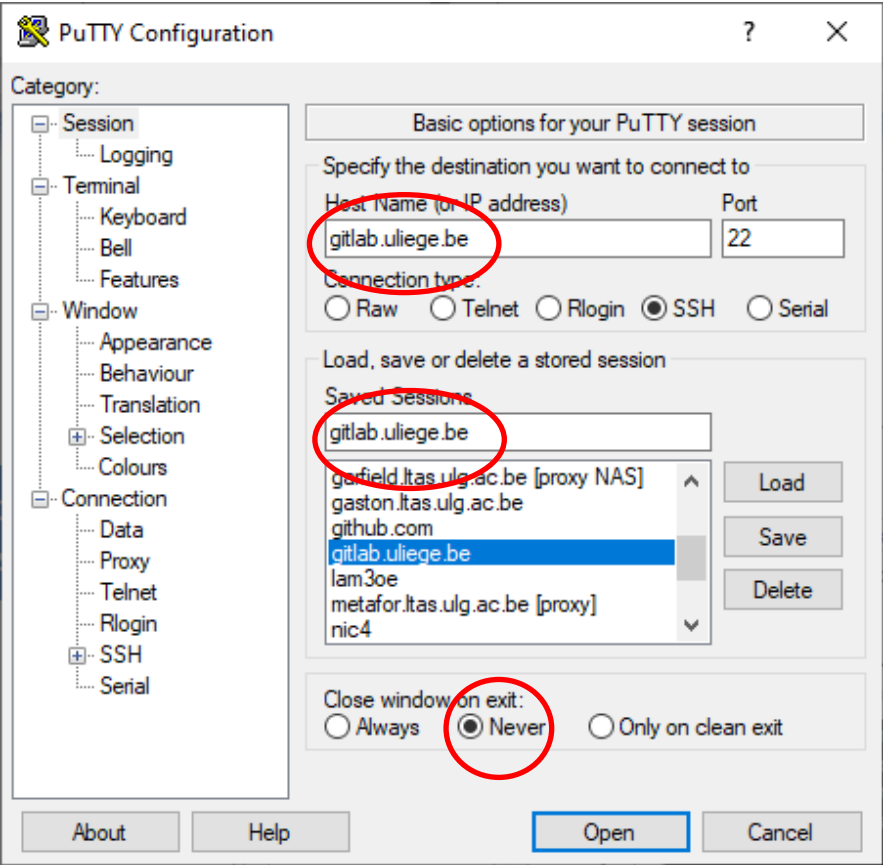
Copiez/collez ce texte  
dans la fenêtre gitlab  
et ajoutez la clef!  
C'est la clef publique.



# FAQ 4 - PUTTY

ATTENTION: A ce stade, n'importe qui possédant votre clef privée peut se faire passer pour vous sur GitLab!

Il reste à configurer PuTTY pour utiliser cette clef:



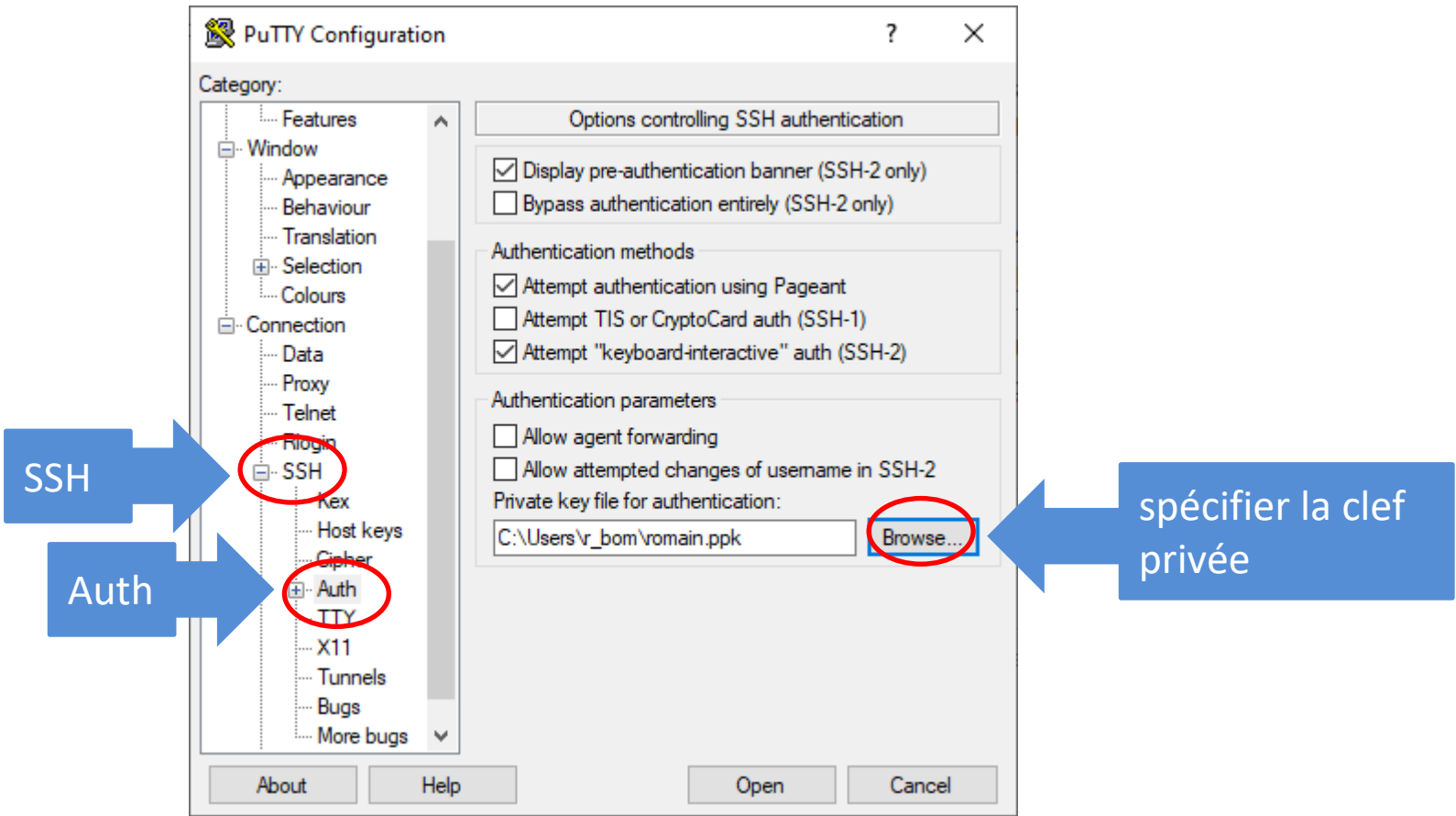
host:  
gitlab.uliege.be

nom de session:  
gitlab.uliege.be

close on exit: Never  
(permettra de tester)

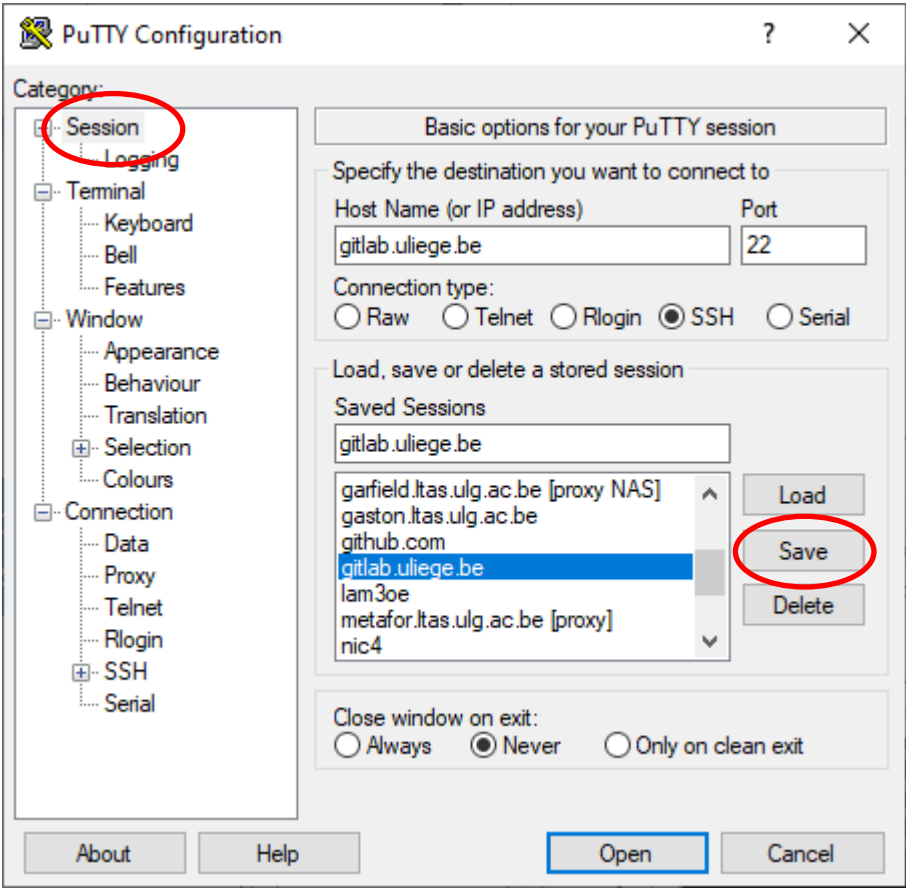
# FAQ 4 - PUTTY

Spécifiez la clef privée dans Connection/SSH/Auth:



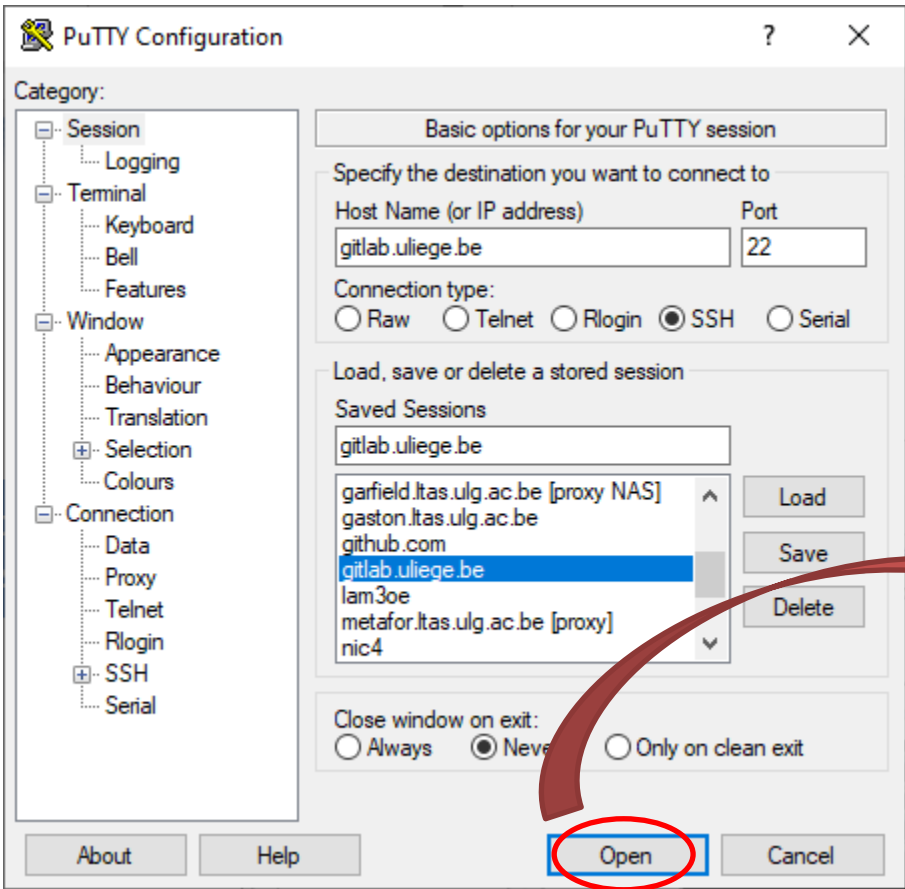
# FAQ 4 - PUTTY

Ne pas oublier de sauver la configuration!



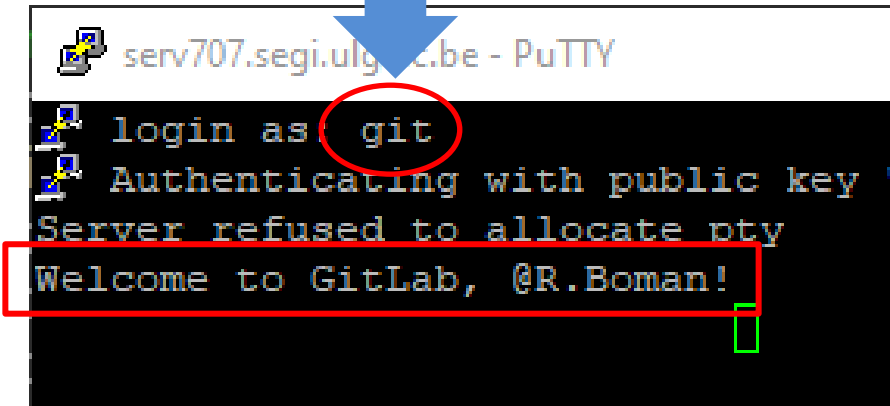
# FAQ 4 - PUTTY

Testez la connexion entre votre PC et GitLab:



Introduisez "git" comme login.

Le message "Welcome to GitLab" indique que votre clef est acceptée!



# FAQ 4 - PUTTY

## Comment ajouter la clé privée de Putty du Gitlab sur les machines Linux?

- PuttyGen -> Conversions -> Export OpenSSH key
- Transférer ce fichier "id\_rsa\_gitlab" dans ~/.ssh
- `chmod 600 ~/.ssh/id_rsa_gitlab`
- `eval `ssh-agent -s``
- `ssh-add ~/.ssh/id_rsa_gitlab`

Voir aussi: <https://gitlab.uliege.be/R.Boman/gmsh-api/wikis/git>

# FAQ 5 - sous-modules

## Git Submodules

Les sous-modules permettent d'inclure des références à d'autres repositories dans un repository donné.

C'est très pratique lorsqu'un projet principal dépend de versions bien précises d'autres projets.

Cela évite de maintenir une trace des différents numéros de versions des sous-projets compatibles à un commit particulier du projet principal.

Cela permet aussi de cloner le projet principal et ses dépendances en une seule commande.

v3493

Assets 4

- Source code (zip)
- Source code (tar.gz)
- Source code (tar.bz2)
- Source code (tar)

Evidence collection

- v3493-evidences-71.json f25801af
- Collected 2 weeks ago

see MR !45

Dependencies

- oo\_nda: v3420
- parasolid: v3035
- linuxbin: v3042
- MetaforSetup: v1258

difficile à maintenir

# FAQ 5 - sous-modules

## Créer un sous-module

Imaginons deux repositories:

- module: le projet principal
- submod: le futur sous-module de module

Pour l'instant il s'agit de 2 projets séparés. On veut ajouter submod à module:

```
git clone git@gitlab.uliege.be:R.Boman/module.git
cd module
git submodule add git@gitlab.uliege.be:R.Boman/submod.git
```

La commande "git submodule add" crée un fichier .gitmodules qui définit la dépendance. submod devient un sous-répertoire de module.

Il suffit alors de commit/push le résultat:

```
git add .
git commit -m "add submod as a submodule"
git push
```

# FAQ 5 - sous-modules

## Cloner un projet contenant un sous-module

Un clone classique ne suffit pas (le sous-répertoire sera vide!).

```
git clone git@gitlab.uliege.be:R.Boman/module.git  
git submodule init  
git submodule update
```

Les 2 dernières commandes

- initialisent le sous-module (dans `.git/config`) et
- font un checkout du commit spécifié dans le projet principal.

Ces 3 commandes peuvent être effectuées en une seule:

```
git clone --recursive git@gitlab.uliege.be:R.Boman/module.git
```



# FAQ 5 - sous-modules

## Qu'est ce qu'un sous-module pour git?

Pour le sous-projet (dans le répertoire module/submod):

- Il s'agit d'un clone tout à fait classique dans lequel on peut travailler comme s'il s'agissait d'un repository normal.
- On peut créer des branches, faire des commits, des pushes, etc.

Pour le projet principal (dans le répertoire module):

- Le sous-module n'est qu'un numéro de commit, rien de plus.
- Ce numéro évolue continuellement avec ce qui se passe dans le sous-module.
- Si on effectue un commit dans le sous-projet, le sous-module sera marqué comme "modified" et un git diff donnera les 2 numéros de version auxquels on s'attend (la version de référence et celle du commit qu'on a créé dans submod).

Par défaut le sous-projet est en "*detached-head state*". Il est donc lié à aucune branche. Pour le modifier, il faut donc tout d'abord créer/sélectionner une branche!

# FAQ 5 - sous-modules

## Merger un projet contenant un sous-module non modifié (1/2)

Le cas le plus simple est celui où le sous-module a été modifié sur origin mais pas localement. On veut donc juste récupérer la nouvelle version du sous-module dans notre branche.

```
git pull
```

**Attention: Le résultat est contre-intuitif!**

- La version de référence du sous-module va être mise à jour (opération invisible).
- La copie de travail du sous-module ne va pas être affectée par cette commande.
- Autrement dit, le sous-module sera affiché comme "modifié" avec un sous-module est "en retard" sur la version d'origin!
- Si, on commit le résultat sans regarder, on supprime la modification d'origin!

# FAQ 5 - sous-modules

## Merger un projet contenant un sous-module non modifié (2/2)

Il est impératif de faire suivre l'opération de merge par la commande

```
git submodule update
```

Cette commande fait un checkout du sous-module correspondant à la version de référence (qui provient ici d'origin depuis le `git pull` précédent).

Même remarque pour un merge de master vers sa propre branche par exemple:

```
git fetch  
git merge origin/master  
git submodule update
```

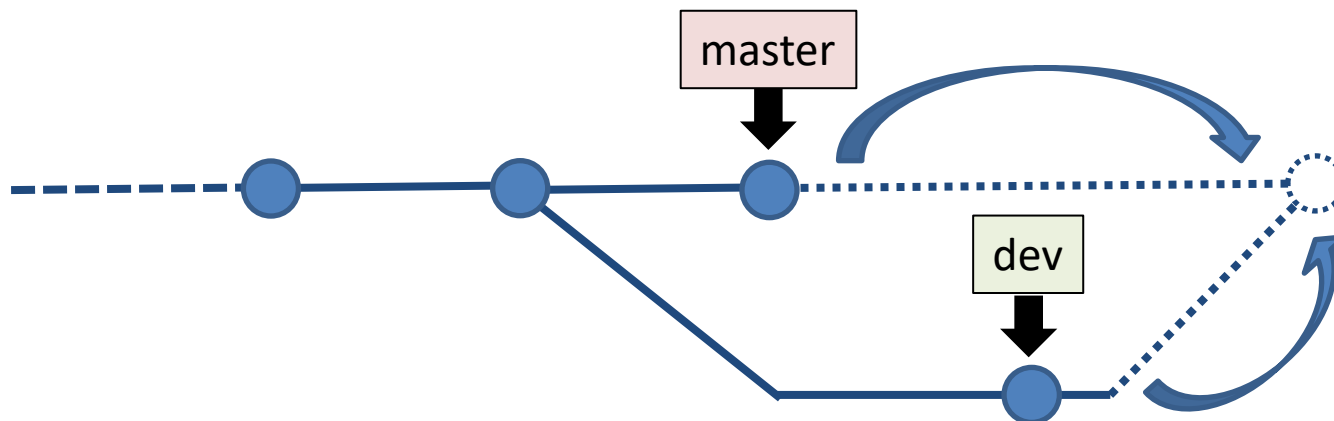
Plus généralement, `git submodule update` rétablit toujours l'état de la copie de travail du sous-module pour qu'elle corresponde à la version de référence. Si des modifications non committées existent, la commande s'interrompt car il y a risque de perte de données.

# FAQ 5 - sous-modules

## Merger un projet contenant un sous-module qui a divergé (1/4)

Imaginons que la branche master fait référence à une autre version du sous-module comme précédemment, mais que un développeur travaillant sur sa propre branche a lui-aussi modifié la version du sous-module.

Si ce développeur veut merger master dans sa branche pour se mettre à jour, il va y avoir inévitablement un conflit qu'il faut résoudre.



# FAQ 5 - sous-modules

## Merger un projet contenant un sous-module qui a divergé (2/4)

Dans le projet principal (répertoire module/)

```
[dans la branche "dev"]
```

```
git fetch
```

```
git merge origin/master
```

```
Failed to merge submodule submod (merge following commits not found)
```

```
Auto-merging submod
```

```
CONFLICT (submodule): Merge conflict in submod
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

# FAQ 5 - sous-modules

## Merger un projet contenant un sous-module qui a divergé (3/4)

On observe les diffs:

```
git diff
diff --cc submodule
index 343da51,e66b9c9..0000000
--- a/submod
+++ b/submod
```

- 343da51 est la version que le développeur a sur disque.
- e66b9c9 est la version qu'il essaye de merger venant de origin/master.

En effet, on peut vérifier dans module/submod:

```
cd submodule
git rev-parse HEAD
343da511e14f73f39d10726a07b82890a1760ad4
```

# FAQ 5 - sous-modules

## Merger un projet contenant un sous-module qui a divergé (4/4)

Il suffit donc de merger manuellement ces versions.

Dans module/submod:

```
git merge e66b9c9
```

Cette commande va créer un nouveau commit de merge, avec résolution des conflits dans le sous-module s'il y en a (c'est une procédure de merge classique).

Enfin, il faut résoudre le conflit au niveau du projet principal:

```
cd ..  
git add submod  
git commit -m "merge master into my branch"  
git push
```

# Utiliser git

```
void mxv(int m, int n, double *a, double *b, double *c, int nbt, int tmax)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        #pragma omp parallel for num_threads(nbt)
```

## Notes

(à intégrer dans ce qui précède)

```
double tstop = omp_get_wtime();
double cpu = tstop-tstart;

OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

std::cout << res;
```



# Notes

## **TODO - A ajouter dans la présentation:**

- Ajouter .gitattributes
- Ajouter .editorconfig
- Ajouter LFS (large file system)
- Note sur l'UTF-8

# Notes

## Comment voir l'info de branch tracking?

```
$ git remote show origin
```

```
* remote origin
```

```
Fetch URL: git@gitlab.uliege.be:R.Boman/CT.git
```

```
Push URL: git@gitlab.uliege.be:R.Boman/CT.git
```

```
HEAD branch: master
```

```
Remote branch:
```

```
master tracked
```

```
Local branch configured for 'git pull':
```

```
master merges with remote master
```

```
Local ref configured for 'git push':
```

```
master pushes to master (local out of date)
```

```
git branch -avv
```

```
* master          40658bb [origin/master] correction of tool kinematics
```


```
remotes/origin/HEAD -> origin/master
```

```
remotes/origin/master 40658bb correction of tool kinematics
```

# Notes

Configuration sous Windows (1x par repository)

```
git config core.filemode input
```



Recommandé:  
les fichiers Unix exécutable  
(chmod u+x) ne seront pas  
considérés "modifiés".

# Fin

```

void mxv(int m, int n, double *a, double *b, double *c, int nbt, int tmax)
{
    #pragma omp parallel for num_threads(nbt)
    for (int i=0; i<m; i++)
    {
        for(int t=0; t<tmax; ++t)
        {
            a[i] = 0.0;
            for (int j=0; j<n; j++)
                a[i] += b[i*n+j]*c[j];
        }
    }

    // loop on thread nb
    int idx2=0;
    for(int nbt=trange.getMin(); nbt<=trange.getMax(); nbt+=trange.getStep())
    {
        idx2++;
        double tstart = omp_get_wtime();
        test.execute(nbt);
        double tstop = omp_get_wtime();
        double cpu = tstop-tstart;

        OMPData res = OMPData(idx1, idx2, siz, nbt, test.getMem(), cpu, test.flops(nbt));

        std::cout << res;
    }
}

```